Vector Addition Systems
and their
Applications in the Verification of Computer Programs

by

Alexander Dixon

Thesis

Submitted to the University of Warwick
in partial fulfilment of the requirements
for admission to the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

November 2022

Contents

List of Tables

List of Figures

Acknowledgments

This work would not have been possible without my wonderful supervisors, Ranko Lazić and Andrzej Murawski. Your phenomenal competence, constructive feedback, and unbounded optimism have been a source of great energy and comfort, and it has been a privilege and an honour to work with both of you. Thank you.

Thank you also to the Department of Computer Science at the University of Warwick, for providing me with a place to work and a phenomenal group of people to work alongside. I am grateful to the Center for Doctoral Training in Urban Science and Progress, who funded my studentship, and to the Engineering and Physical Sciences Research Council for the same.

Thank you to those who I have collaborated with on works over the years, not least Igor Walukiewicz, for your invaluable insights. Thanks also to those members of staff at the University of Warwick who have offered advice and guidance throughout my time as a student.

Thank you to my friends, near and far, for the support which you have shown, and for putting up with me for far longer than I ever would have. A particular thanks to Sam, Finnbar, and Rhiannon, for sharing in the highs and lows of postgraduate life, and a huge thank you to my dear friend Andrew, without whom I would have gone mad long ago.

Thank you to my family, especially my parents, brother, and grandparents, for your unwavering love and kindness, and for asking how my thesis was coming along the bare minimum number of times.

Finally, I would like to thank my darling husband, Jack. I am a better person for knowing you, and I love you with all my heart.

For Jack

Declarations

This thesis is wholly the author's own work, except where work is based on collaborative research, in which case the nature and extent of the author's contribution is indicated. This thesis has not been submitted for a degree at any other university.

1 Publications

Parts of this thesis have been previously published by the author in the following works.

[40] Alex Dixon and Ranko Lazic. KReach: A tool for reachability in Petri nets. In *Proceedings of TACAS*, volume 12078 of *LNCS*, pages 405–412. Springer, 2020. doi:10.1007/978-3-030-45190-5_22

[41] Alex Dixon, Ranko Lazic, Andrzej S. Murawski, and Igor Walukiewicz. Leafy automata for higher-order concurrency. In *Proceedings of FOSSACS*, volume 12650 of *LNCS*, pages 184–204. Springer, 2021. doi:10.1007/978-3-030-71995-1_10

[42] Alex Dixon, Ranko Lazic, Andrzej S. Murawski, and Igor Walukiewicz. Verifying higher-order concurrency with data automata. In *Proceedings of LICS*. IEEE, 2021. doi:10.1109/LICS52264.2021.9470691

2 Funding

This research was funded by the Centre for Doctoral Training in Urban Science and Progress. The CDT is supported by the University of Warwick and by EPSRC Grant Number EP/L016400/1.

Abstract

Vector Addition Systems (and, equivalently, Petri nets) are a widespread formalism for modelling across a spectrum of problem domains, from logistics to hardware simulation. In this thesis, we firstly explore two classic decidability problems for these models: *reachability*, whether one can get to a given configuration, and *coverability*, whether one can exceed it. These problems are sufficent to express a wide class of verification properties for models derived from real-world use cases, including safety and deadlock-freeness. We present and implement a number of approaches for solving both the coverability and reachability problems, including `KReach`, the first known implementation of a complete decider for the general Petri net reachability problem.

Petri nets offer a natural model of concurrent processes and one of the most common modern use cases for the model is in the verification of safety properties for software, especially sofware with concurrency. In the later half of this work we address some approaches to deciding properties of programs written in Finitary Idealized Concurrent Algol (`FICA`), a prototypical language combining functional, imperative, and higher-order concurrent programming. We introduce a new family of "leafy" automata models, all based on a novel representation of internal configurations as a tree structure whose semantics is inspired by game-semantic interpretations of `FICA` terms. We give translations from such terms to our automata and across the work derive decidability of some useful properties for successively more expressive subsets of terms, using a variety of methods including via reachability on Petri nets. We believe these models will help to unify the game- and automata-theoretic views of programming languages and provide a useful basis on which to further study the theory of concurrency.

Chapter 1

Introduction

The Petri net is arguably older than the field of theoretical computer science itself. Petri nets were originally borne not out of the study of mathematical objects, but as a descriptor of chemical catalytic processes [133, Foreword by C.A. Petri]. Treating Petri nets as a representation of *communication*, and not just arbitrary domain-specific processes, was not a consideration until much later [125]. It took some time [75] to determine that Petri nets are equivalent to a parallel programming schema called Vector Addition Systems [87]. Research into Petri nets and VASs has not slowed in the 80 years since their original formulation; foundational results about Petri nets are still being discovered, with the complexity of cornerstone decision problems being closed as recently as 2021 [30, 102].

It should come as little surprise, then, that Petri nets have a storied history as a formalism for modelling real-world processes. The expressiveness of the model allows for the representation of everything from business process modelling [36] to systemic bug reporting [35]. It is for precisely this reason that Petri nets and Vector Addition Systems are seen as a viable and fruitful area of study. By producing tools that can check properties of Vector Addition Systems, we enable the use of Petri nets to not only model but also verify properties like safety and the avoidance of deadlocks in a plethora of problem domains.

Programming is the mechanism by which humans set computers to work. If we wish for two or more agents to share resources—be that communicating systems on a network, or multiple threads on the same machine, or anything else—we enter the world of concurrent and parallel computing and, when that concurrency is systematised, concurrent and parallel programming. Petri nets and Vector Addition Systems are a natural fit for modelling these concurrent computations.

Concurrent and parallel programming has been the subject of study in a number of subdomains within theoretical computer science. Petri nets/VAS are a natural fit from an automata-theoretic perspective. In the world of programming language theory, entirely new subdomains have been developed to help us understand the implications of concurrent systems. For example, in game semantics (the subject of Section 7.2), evaluation of concurrent terms is modelled as a game played between two players, the term itself and the environment in which the term is being evaluated [70, 93]. Whether the theory of game semantics can be fully unified with the theory of automata remains to be seen.

This thesis can be seen as an homage to Petri nets and Vector Addition Systems as a tool for the systemic treatment of complex, concurrent, real-world systems. In the first part of this work we shall look at the cornerstone problems to which the analysis of properties of such systems can be reduced, and present a number of new tools that allow for exactly that kind of analysis. Later we shall dive into the application for which Petri nets have been most widely studied: the modelling of parallel and concurrent programs. We look at the game-semantic view of these systems and ally it with our own automata-theoretic encoding.

1.1 Thesis Outline

In Chapter 2 we lay out the history of Petri nets and Vector Addition Systems: where they come from, what we can do with them, and what has been discovered about their properties. We introduce the important results that act as milestones in the study of Petri nets and Vector Addition Systems, and give a sampling of some of the problem domains where Petri nets have been used as a modelling tool.

Chapter 3 introduces formalisms that will be used throughout the remainder of the work, in particular both algebraic and diagrammatic representations of Petri nets, Vector Addition Systems, and Vector Addition Systems with States. We shall also show how these three are equivalent, and introduce some more concepts and notation that will prove useful in further chapters.

In Chapter 4 we focus on the *coverability problem* for Vector Addition Systems with States, that asks: can we reach a configuration that is at least some particular size? We outline a number of approaches by which that question can be answered, and walk the reader through an implementation of one such approach.

In Chapter 5 we turn our attention to a much more thorny problem, the *reachability problem*, that asks only whether we can reach some specific configuration. We describe the mechanism by which it was originally proven that this problem can be decided, and introduce a tool that implements precisely that mechanism. We also show how that tool can be used to decide instances of the coverability problem.

Chapter 6 introduces a new form of automaton, the *leafy automaton*, that maintains an internal tree that grows and shrinks as it reads in a word over a particular infinite alphabet. We show that this is capable of representing properties of a prototypical programming language called (Finitary) Idealized Concurrent Algol via game semantics in Chapter 7.

In Chapter 8 we devise new ways of restricting undesirable forms of communication within the tree. The variant presented, so-called "local leafy automata", impose a restriction on the length of updates when adding or removing leaves. We show that this corresponds to a particular fragment of `FICA` where variables are not allowed to appear more than two free function calls away from their definition. This allows us to recover decidability of decision problems for a large fragment of the `FICA` language, though some constructs (in particular unbounded iteration) are still forbidden.

In Chapter 9 we create an alternative variant of leafy automata where control states and memory values are divided and treated separately. This inspires a notion of idempotence, through which we show decidability of emptiness on a fragment of `FICA` that includes every programming construct and excludes only particular (so-called "deep") uses of semaphores.

We shall conclude the thesis by revisiting the principal contributions, identifying potential for future work and any questions left open, and sharing insights into some of the work that is already underway.

Chapter 2

Background

The history of research into Petri nets and Vector Addition Systems is long and storied—a superficial inspection of the corpus[1] indicates well over 200,000 relevant pieces of literature. To try to present the fullness of the branching, deeply intertwined and still-ongoing research within this document would be a Herculean, if not Sisyphean, task. Instead, we shall present in some detail the "critical path" of Petri net and Vector Addition System research taking us from the original introduction of the models to this work.

Over the years, Petri nets have gone by a number of different names. In the latter work of Petri himself they were referred to as *special nets* in contrast with the many extensions and variants of the model for which he coined the umbrella term *general nets*. In some works [80, 142] they have been called *place/transition nets*. We shall exclusively use the term Petri net to refer to the classic form as laid out in Section 3.5, with no additional augmentations or restrictions.

2.1 Early Petri nets and Vector Addition Systems

In 1939, Carl Adam Petri wished for a diagrammatic system by which he could visualise chemical processes [133, Foreword]. In particular, he wanted to model the way in which some number of chemical compounds could react in the presence of a catalyst. To this end, he devised a representation whereby the populations of given chemicals were located in places, and transitions could move a specific number of each type of compound from one place to another, hence indicating that a reaction had occurred.

[1]Read: a Google Scholar search for "Petri nets".

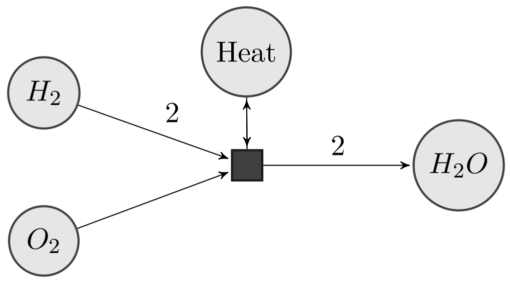


Figure 2.1: A Petri net which might represent a condensation reaction (the creation of water) from Hydrogen and Oxygen in the presence of a heat source.

Owing to a burgeoning academic interest in computational models, Petri's seminal presentation of his nets in his thesis *Kommunikation mit Automaten* [125] introduced the formalism not as a visual system but as a mathematical object, with (the imposition or relaxation of) causality as the primary setting for their study. It was not until later that the natural modelling capabilities across a litany of concurrent problem domains were rediscovered, in part by Holt and associates [78, 134].

Almost concurrent with the work of Holt et al., Karp and Miller introduced an alternative schema for the representation of parallel processes, which they called Vector Addition Systems (VAS) [87]. That work already introduced a formulation of the reachability and coverability problems for VAS, and gave a finite construction for a tree representing coverable vectors, a presentation of which is given in Section 4.3.1.

A decade later in 1979, Hopcroft and Pansiot [79] generalised the Vector Addition System model to include a notion of control states, which impose an additional restriction on when transitions may be fired. Moreover they showed that VAS and VASS are equally expressive up to the dimensionality of vectors (see Theorem 3.1).

It was readily observed that Petri nets and Vector Addition Systems were equivalent. In between the introduction of VAS and VASS, Hack [75] formalised this notion; and so all three models (VAS, VASS and Petri nets) have been considered pairwise equivalent since. We will see precisely how the relationships between the three can be formulated in Chapter 3. In what follows, we shall intentionally confuse Petri nets and Vector Addition Systems in the interest of presenting a single coherent timeline of results over both models.

2.2 Decision Problems

The complexity and decidability of problems over Petri nets have long been a matter of study, with the most long-standing question—the complexity of the reachability problem—having finally been closed during the production of this thesis. The formal definitions of the coverability and reachability problems are given as Definitions 4.1 and 5.1 respectively; each forms the focus of its respective chapter.

2.2.1 The Coverability Problem

The coverability problem was one of the first decision problems for Petri nets to be shown to be decidable. As noted, Karp and Miller's coverability tree construction [87] is itself a witness of decidability of the coverability problem, since the algorithm they give for constructing the tree is guaranteed to terminate. Some time later, Cardoza, Lipton and Meyer [24] and Lipton [120] showed that the coverability problem requires exponential space, and thus, by reduction to coverability (via a mechanism described in Proposition 5.1), the reachability problem inherited an exponential space lower bound. Two years later, Rackoff gave a proof [128, Theorem 3.5] that the coverability problem can be solved in space at most $2^{cn \log n}$ nondeterministically—by showing that there is an upper bound on the maximum length of some covering run, simply checking all such runs constitutes a terminating algorithm. Rackoff's work closed the complexity of coverability; the problem is complete for EXPSPACE.

As computers have become exponentially more powerful, the once-scary EXPSPACE bound has become less daunting, and numerous successful attempts have been made to solve the coverability problem in real-world settings. Most commonly, coverability is approached by either forward or backward search. In forward search one computes the set of coverable markings from an initial marking, and checks (at appropriate times) whether the target falls within that set. The Karp-Miller construction is the prototypical implementation of such a forward search, and many attempts have been made to hone the approach [50, 55, 62, 126, 137, 144]. These works revolve around minimising redundant computation while constructing the coverability tree (or, in some cases, the coverability set), including by pruning parts which are guaranteed not to expand the set of covered markings.

Conversely, backward algorithms start from the target marking and seek to include the initial marking in the set of markings which *can* cover the target. This technique revolves around the notion of well-structured transition systems (see Section 3.3); the prototypical backward coverability algorithm (Algorithm 2) has been shown to take doubly exponential time [20]. Much like forward search, backward search is accelerated by pruning. The pruning criteria are typically derived either from relaxations of the coverability problem [13] or from static analysis of the nets [63]. Beyond search, a number of effective criteria for coverability have been derived in integer programming, and can be used to quickly rule out coverability with the help of SMT solvers either statically or with iterative refinement of criteria [45, 47, 148]. In Section 4.3 we discuss and implement some of the more recent approaches for solving coverability, incorporating a number of these techniques.

2.2.2 The Reachability Problem

Complexity of the reachability problem, in contrast to coverability, proved to be a significant challenge for many years. That reachability was even decidable was not claimed until Sacerdote and Tenney [140] in 1977. However, they did not include a formal proof of the claim. It was not until the 1981 work of Mayr [107] that we get such a proof; work of Kosaraju published the following year [90] offered a simplified flavour of Mayr's proof. (The presentation of Kosaraju is the focus of Chapter 5, in which we explain, implement and test Kosaraju's algorithm on real nets.) Kosaraju's and Mayr's presentations of decidability are infamously involved[2]. A book of Reutenauer [136] exposes the algorithm in magnificent detail. That book has since been translated into English [135]. Three decades later, Lasota [96] revisited the proof to offer an intuitive and high-level overview of the algorithm. Lambert [95] refined and simplified Kosaraju's presentation via a new structure called the "marked graph-transition sequence", an alternative treatment of Petri nets with initial and final markings. In modern parlance, the combined efforts of Sacerdote and Tenney, Mayr, Kosaraju and Lambert are colloquially known as the *KLMST* algorithm [103].

As of 1982, there was no known upper bound, and it was only known that the KLMST approach required non-primitive-recursive space [114]. Astonishingly, it took another three decades before this state of affairs changed. In 2015, Leroux and Schmitz [103] gave the first ever upper bound for reachability in general Petri nets. In making use of then-recent results for termination in well-structured transition systems (an abstraction into which Petri nets naturally fit, see Section 3.3), Leroux and Schmitz were able to show that the general reachability problem could be solved in cubic-Ackermann time (see Section 3.1.2), by showing that these well-structuring properties also hold for the decompositions induced by the KLMST algorithm. Not long after, Leroux and Schmitz further improved on this result [104] by offering significant refinements of the KLMST algorithm to reduce their earlier upper bound from cubic-Ackermann to regular Ackermann, and moreover introduced a primitive-recursive upper bound if the dimension is fixed.

[2]As devastatingly put by Lambert [95, Introduction]: "The complexity of the two proofs (especially in [107]) wrapped the result in mystery."

In between gradual refinements of the KLMST algorthm, Leroux [83] offered an alternative proof of decidability using an unrelated scheme, namely the enumeration and refinement of invariants defined in Presburger arithmetic, with support of the marked graph sequences of Lambert [95].

Iterating on the lower bounds for reachability proved to be an even tougher challenge than the upper bounds. Lipton offered the first EXPSPACE lower bound for reachability in 1976 [120]. From that point, it was long conjectured unofficially that the problem may be complete for EXPSPACE, and another forty years passed before this was shown to not be the case. In 2019, work of Czerwiński et al. [31] showed that the reachability problem was not elementary, and more precisely that it was at least TOWER-hard (See Section 3.1.2). This represented a significant step forward, obsoleting or re-proving a number of results in more expressive formalisms [98, 99] and refuting attempts to solve the problem more efficiently [19]. The TOWER lower bound of [31] is earned by showing that particular so-called "amplifier" gadgets (sub-nets) can be constructed which enforce a factorial relationship between the number of tokens in some places.

The bounds on the reachability problem snapped shut in 2021, when Leroux [102] and Czerwiński and Orlikowski [30] independently and almost-simultaneously[3] proved that the reachability problem for Petri nets is not primitive recursive. The two proofs proceed by different methods, though in both cases the technique revolves around increasing the power of the amplifying gadgets described in the non-elementary proof of Czerwiński et al. [31]. The work of Leroux introduces several gadgets which, taken together, produce an Ackermannian amplification. Czerwiński and Orlikowski offer a precise result [30, Theorem 1]—that, for every $k > 3$, the reachability problem in dimension $6k$ is hard for $\mathcal{F}_k$ [4]—by recursively building a similar family of gadgets to those of Leroux. ACKERMANN-hardness follows as a corollary. Both of these results, taken with the upper bound of Leroux and Schmitz [104], give an exact complexity of ACKERMANN, and the problem is now closed in the general case. Very recent work of Lasota [97] simplifies the construction of Czerwiński and Orlikowski, and in doing so improves the lower bound in fixed dimension from $\mathcal{F}_k$ in dimesnion $6k$ to $\mathcal{F}_k$ in dimension $3k + 2$.

As we shall discuss in Chapter 5, the long hiatus of upper bound results was not conducive to attempts to solve the problem in practice. Prior to the work of this author [40] no implementations of any reachability algorithm are known. Very recently, works of Amat, Zilio and Hujsa [6] and Blondin, Haase and Offtermatt [15] offer implementations of semi-decision procedures

[3]The proof of Czerwiński and Orlikowski was published online only two days after that of Leroux.

[4]$\mathcal{F}$ is the hierarchy of fast-growing functions, as defined in Section 3.1.2.

underpinned by heuristic-driven search, which show encouraging results and will hopefully spur on further attempts to compute reachability for nets arising from real-world systems, even if the general problem remains intractable.

Other Decision Problems Due to their relative expressiveness, many standard automata questions are immediately undecidable on Petri nets. For example, Jančar [81] showed that trace equivalence and reachability set equality are both undecidable, via undecidability of bisimilarity. It is sometimes easier to solve so-called *structural* variants of Petri net problems than their specific counterparts, where a structural variant of some property holds for any possible initial marking. For example, while boundedness (finiteness of the set of reachable markings) is decided by reduction to coverability [120], *structural* boundedness of a net can be decided in polynomial time by reduction to integer linear programming [109]. Survey works due to Esparza [44] and Esparza and Nielsen [46] expose the full scope of such problems and their decidability results (as known up to their respective dates of publication).

2.3 Applications of Petri nets

Petri nets are a natural fit for modelling and checking properties of a wide variety of systems. It is not uncommon for modelling literature to introduce a new Petri net formalism which happens to be a perfect fit for their setting; some such extensions of Petri nets will be discussed in Section 2.4. Here we shall mostly interest ourselves with applications of standard Petri nets with no extensions (or only extensions which can be simulated by Petri nets themselves).

Chemistry As intimated previously, the Petri net was originally conceived in order to model chemical processes [133, Foreword]. Since their invention, Petri nets have continued to be used for the modelling of chemical processes. Chemical reaction networks [127] offer a very similar treatment of chemical processes to Petri nets: the network places compounds and reactions on vertices and uses weighted directed arcs to connect compounds and reactions (usually expressed via an incidence matrix) [5]. With no further treatment, chemical reaction networks are isomorphic to Vector Addition Systems and can therefore be expressed as, and modelled using techniques for, VAS. (It is not uncommon for chemists to wish to investigate the dynamical properties of chemical processes, for which alternative formalisms like population protocols [8] or augmentations of chemical reaction networks [73] have been adopted.)

Biology and Biochemistry Biological applications of Petri nets are often imported from chemistry. For example, the modelling of metabolic pathways [77, 130, 131] is inherently biochemical in nature. Quantitative modelling of biochemical processes uses exactly the same analogues between features of the system and of nets [10, Section 4.1]: compounds and catalysts (here, enzymes) are represented by places and the presence/quantity thereof represented by tokens; transitions denote the reactions and interactions between compounds in the presence of any catalysing agents. Applications in biology do extend beyond biochemistry, though: Petri net models have been produced for a number of specific biological domains, including disease prognosis [123].

Business Process Modelling Modelling business processes has been provocatively referred to as the "killer app" for the formalism [145]. Business processes do not have a precise definition, though there are some formal objects that attempt to capture business processes, including UML activity diagrams [16] and the Business Process Model and Notation [74]. Both of these are visually akin to flowcharts and are designed to be both precise (for modelling) and intuitive (for non-technical users). Figure 2.2 gives a visual representation of a Petri net, the manual firing of transitions in which might be imagined as a business process in itself. The role of tokens in a Petri net corresponding to some operational process is not fixed, though it typically means the fulfilment of some condition or the presence of some agent in a particular state. (Figure 2.2 includes instances of tokens in both roles.) One notable instantiation of Petri nets for business processes is in VERIFAS [36], a tool which can verify properties of workflows by exploiting the Karp-Miller coverability procedure for Petri nets.

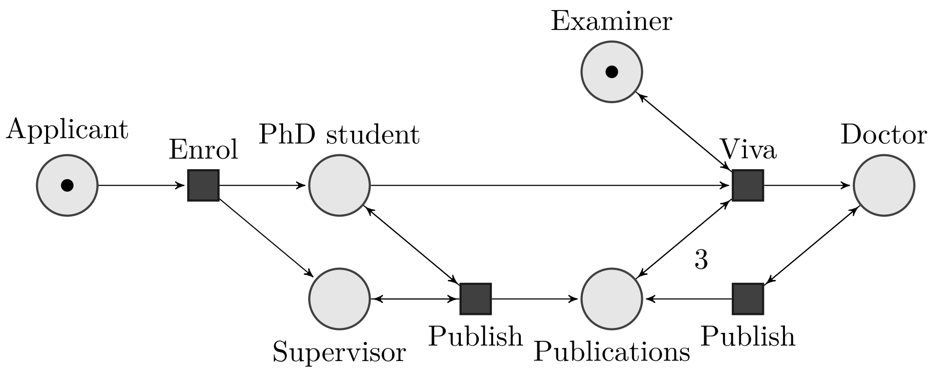


Figure 2.2: A representation of the postgraduate life cycle, using a Petri net as an analogue for business process modelling. Temporal flow is approximately left-to-right.

The VERIFAS technique of [36] revolves around modelling a variant of linear temporal logic specific to the needs of their encoding of processes. In fact, Petri nets are useful for verifying temporal properties of concurrent systems more generally. German and Sistla [65] give a theoretic treatment of this use case by showing that Vector Addition Systems with States can verify whether some specification provided in so-called *propositional linear temporal logic* (PTL) can be satisfied in runs of computations with unboundedly many concurrent processes. With a more practical bent, Kaiser, Kroening and Wahl [85] consider the automated verification of concurrent processes whose concurrency is defined by *pthreads*, a common architecture- and operating system-independent parallel execution model [23]. The tool that Kaiser, Kroening and Wahl present, `BFC`, is highly optimised for checking coverability in those models—it consistently outperforms other tools on coverable nets—but is nevertheless competitive across the gamut of instances in the literature; `BFC`, and the instances produced by their modelling, are now part of the pantheon of classic tools and benchmarks against which new coverability implementations are often evaluated [13, 63].

Transportation and Logistics Petri nets have also been shown as applicable to the modelling and verification of transportation networks. Such systems are naturally discrete, and involve the pointwise interaction of concurrent, fungible agents. Detecting the potential for gridlock—the inability for any vehicle to make progress—is exactly analogous to the property of *(structural) liveness*, which is reducible to checking reachability [115]. Petri nets have similarly been used model the logistics and transportation of fungible resources [26].

New Applications The search for new applications for Petri net modelling is ongoing. The annual Model Checking Contest [1, 7] features a call for models, in which Petri net community members are encouraged to submit exemplar models both of academic and industrial interest. Models in the corpus range from business process modelling of swimming pool facilities[5] and railways[6] to verifying coherence in processor caches[7] and, most recently, safety for autonomous vehicles[8]. Such models are typically derived from programmed systems and are often parameterised by some variable such as the number of agents or the scale of the system. The size of models varies from tens of places and transitions to millions.

[5]https://mcc.lip6.fr/pdf/SwimmingPool-form.pdf
[6]https://mcc.lip6.fr/pdf/CircularTrains-form.pdf
[7]https://mcc.lip6.fr/pdf/ARMCacheCoherence-form.pdf
[8]https://mcc.lip6.fr/pdf/AutonomousCar-form.pdf

2.4 Extensions of Petri nets

We shall briefly highlight some extensions of interest that have cropped up over the years, and some examples of how they have been shown to be fruitful areas of study.

Coloured Petri nets The inability for tokens to be imbued with information often proves troublesome when trying to model systems in which entities are not completely fungible. A common extension to Petri nets is to allow tokens to be *coloured*, which can be considered synonymous with *tagged* or possibly *typed* [64, Section 3]. Along with the standard encoding of places and transitions, the presence of colours allows the transitions of a net to differentiate between tokens. (As an aside, if the set of colours is finite, then coloured nets are as expressive as Petri nets, though the dimension of an equivalent Petri net may be much larger.) First introduced by Jensen in 1986 [82], coloured Petri nets have been used in a variety of settings, from processor hardware modelling [22] to air pollution control via traffic management [37]. Of particular relevance to this thesis, an implementation of (a flavour of) coloured Petri nets exists in the Haskell programming language [132].

Time-dependent Petri nets In a timed Petri net, transitions take some amount of time to fire. Depending on the choice of terms, this time may be fixed [149] or range over some interval [124]. Hence the firing of transitions is constrained; queries may be posed about whether some action can occur within some minimum or maximum time span. This is of particular value when the time taken to achieve a task matters, such as in systems whose speed is meaningfully bounded by the speed of light such as programmable logic units [101], or slow-moving logistical systems like international freight [43]. Other time-dependent Petri net models include Time Petri Nets, first introduced by Merlin [110].

Branching VASS Branching VASS (BVASS) are a variant of VASS which include two flavours of transition. The formalism was first so named by Verma and Goubalt-Larrecq [147], though models with similar semantics had been seen prior [129, 146]. As well as the classic state-state transition which modifies the vector as in VASS, BVASS include a transition which splits one configuration into two. The descendants each end up in some new state (as dictated by the transition) and the vectors of the descendants can be split arbitrarily so long as they sum to the vector of the antecedent configuration. Hence a run in a BVASS is no longer a path, but a *tree* labelled by configurations. Being at least as expressive as VASS, BVASS inherit the lower bounds for their corresponding coverability, boundedness and reachability problems (among others). It was shown by Demri et al. [32] that the former two are complete for doubly exponential time; despite promising recent works [28, 108] the general reachability problem remains open. BVASS have proven useful both for theoretic and real-world modelling, bearing relationships with timed automata [28], recursively parallel programs [18], and databases [17].

*   *   *

Let us turn our attention to the ancestry of the latter half of the thesis—Chapters 6, 7, 8 and 9, which focus on the verification of properties of computer programs.

2.5 ALGOL and FICA

The programming language FICA is the focus of Chapter 7, in which the language is introduced in detail. FICA, which is short for Finitary Idealized Concurrent ALGOL, is a prototypical language which features support for a variety of programming paradigms including imperative, functional, and higher-order concurrent programming. It therefore serves as an interesting testbed for verification techniques for programming language properties, and especially in settings which treat concurrency as first-class (including those featured in this thesis).

The history of FICA, unsurprisingly, begins with ALGOL. It is hard to overstate the impact that ALGOL has had on programming, both as an engineering profession and as a theoretical endeavour. ALGOL has underpinned programming language design since the mid-20th century, and is the progenitor of many programming constructs that are ubiquitous across languages today. It would be foolish to try and map out the complete ALGOL extended universe here; we shall only discuss the branch of the family tree that leads us to FICA.

ALGOL, short for **ALGO**rithmic **L**anguage, first appeared as ALGOL58 as an attempt to provide a unified language for expression of algorithms between the *Association for Computing Machinery* (ACM) and the *Gesellschaft für Angewandte Mathematik und Mechanik* (GAMM) [9]. The language was a compromise design and was quickly superseded by ALGOL60, which rose to supremacy among theoreticians as the language of choice for algorithmic programming.

In an chapter of *ALGOL-like Languages* in 1981, Reynolds [138] lauds the original design of ALGOL60 for its prioritisation of high-level semantics over efficient but low-level routines. To embody the traits that Reynolds viewed as essential to ALGOL, he introduced a distilled, high-level language called Idealized ALGOL, or IA.

Reynolds' claim was this:

> Algol is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus. [138, page 3]

The formalisation of IA given by Reynolds is exactly that: a unification of standard imperative programming constructs and a typed lambda calculus with call-by-name semantics.

The formal semantics of a parallel ALGOL-like language were first introduced at the turn of the millenium with Parallel Algol [21], due to Brookes. In that work, a blocking `await` construct controls threaded access to critical sections of code.

In [70], concurrency was introduced to IA in a similar vein to Brookes's Parallel Algol. Here the authors introduce parallel composition of statements (||) with binary semaphores controlled via fine-grained **grab**() and **release**() primitives. The point of this modified language, now Idealized Concurrent ALGOL (ICA), was to facilitate analysis of the language through game semantics, which is naturally conducive to modelling concurrency. Indeed, the game model for concurrency described in [70] is simpler than its sequential counterpart. We shall discuss the game-semantic concerns of ICA later in this chapter.

It is in [70], due to Ghica and Murawski, that we first encounter FICA proper. In this article, ICA is restricted to a single, finitary data type, hence becoming *Finitary* ICA. The purpose of the restriction to a finitary datatype was to find a context in which computer-aided verification (in particular for [71], (may)-equivalence) of the language could be rendered decidable. We continue to make progress on that same goal in the remainder of this work.

2.6 Game Semantics

Game semantics (to be explored more concretely in Chapter 7) is a model by which the evaluation of a term is rendered as a game between the term itself and the evironment in which it is being evaluated [3, 118]. It has been described by Harmer and McCusker as a middle-ground between models of concurrency and domain-theoretic semantics [76].

Game semantics have long been a means for verifying properties of ALGOL derivatives. In 1996, Abramsky and McCusker put forth a game semantics for IA (one of the precursors to FICA, without concurrency) [4]. Malacaria and Hankin successfully made use of these semantics to present an interpretation of programs in terms of data flow [106]. While the model of Abramsky and McCusker is fully abstract—that is, it correctly captures all operational equivalences between programs—it is relatively complex and reasoning about the model proved to be challenging [66, Introduction].

At the turn of the century, Ghica and McCusker showed that some meaningful portions of the game semantics of programming languages like IA can be captured by simple automata such as an extended form of regular expressions [66]. This result led to a flurry of research into the use of game semantics for program analysis and verification. Follow-up work of Ghica and McCusker [67] showed that, considering at most second-order terms, the game semantics of a fragment of IA can be captured with fully abstraction using only regular expressions. Dimovski and Lazić [39] gave an alternative translation of the game semantics of a similar second-order fragment into sentences of the Communicating Sequential Process (CSP) algebra, properties of which were then able to be checked directly using pre-existing tooling [105]. Contextual equivalence of the third-order fragment was shown to be decidable by Ong [121] by representing the corresponding game semantics with deterministic pushdown automata (and hence inheriting decidability from the decidability of equality for deterministic context-free languages). Furthermore, Murawski and Walukiewicz [119] showed in 2005 that contextual equivalence of the third-order fragment is still decidable in the presence of iteration. By this time, the fourth-order fragment had already been shown undecidable by Murawski [116].

Given their natural applications to concurrent programming, it is a curiosity that game semantics was initially constrained to the sequential universe. The first occurrence of a game semantics for a concurrent flavour of Algol was due to Abramsky [2] and was an attempt to model the Parallel Algol of Brookes [21]. Laird gave a game-semantic account of CSP using an interleaved representation of concurrency [92], making use of a raft of additional structure to enforce appropriate constraints on the representation. Later, Ghica and Murawski [70] introduced Idealized Concurrent Algol and with it a fully abstract game semantics, again based around interleavings of concurrent processes. Further work of Ghica, Murawski and Ong [69] builds out a variant of the same, in which the number of concurrent threads is embedded in the type system. In this variant, called Syntactic Control of Concurrency (SCC), their (fully abstract) game semantics is regular, even when iteration is allowed, and thus bounded contextual equivalence of this fragment becomes decidable. The same work offered some results on the contextual equivalence of corresponding fragments of `FICA`: for the second-order fragment it is undecidable, but for the first-order fragment it is decidable (moreover, that fragment is regular, because there is no opportunity for unbounded behaviours to occur). Further work by Ghica and Murawski [68] gives a translation from SCC to CSP.

It is worth noting that earlier approaches such as that of Murawski and Ong explicitly require a bound to be enforced (syntactically in the case of SCC) on the amount of concurrency allowed. The work of this thesis seeks to identify fragments of `FICA` which include *unbounded* concurrency and *unbounded* order of terms, for which some interesting problems—if not equivalence—can be shown to be decidable.

The work of this thesis follows an approach to games based on constrained interleavings of moves. Another popular approach to game semantics involves event structures [25], which are used to capture dependencies between moves via explicit partial orders. This approach belongs to the so-called "truly concurrent" tradition in modelling. Investigating the potential for a truly concurrent approach to inspire verification routines for `FICA` is an interesting topic for potential future work.

Chapter 3

Preliminaries

In this chapter we shall lay a foundation and define terms and conventions which shall reappear throughout the remainder of the work. Most critically, we shall define the Petri net, the Vector Addition System, and the Vector Addition System with States, and formally define the relationship between these three formalisms.

Not all of the terminology, notation, or syntactic conveniences for the thesis are defined here. In cases where such definitions are only going to be used in one chapter or section, they will be given in the appropriate chapter or section. Here we define only those that will reappear throughout the work.

3.1 Mathematical Notation

In the greatest mathematical tradition, we shall use uppercase blackboard letters to represent sets of numbers. In particular:

- $\mathbb{Z}$ is the set of integers;
- $\mathbb{N}$ and $\mathbb{N}_+$ are the sets of nonnegative integers and positive integers, respectively (unless specified, the "set of naturals" is taken to mean $\mathbb{N}$);
- $\mathbb{Q}$ is the set of rational numbers; and
- $\mathbb{R}$ is the set of real numbers.

3.1.1 Sets, Sequences and Functions

Notation for sets and sequences is standard in the literature, with sets denoted by {braces} and sequences by (parentheses). We shall write $\cup$ for set union, and $\uplus$ when it is a given that the two operands are disjoint. Usage of $\uplus$ may be descriptive (naturally arising from previous definitions) or prescriptive (a statement the disjointness is a prerequisite for correctness). Any prescriptive uses of $\uplus$ can be enforced by tagging the elements of the first and second operand with $_1$ and $_2$ respectively. The shorthand $\{x, \ldots, y\}$ is the set of integers from x to y inclusive.

We take a (binary) relation $R \subseteq A \times B$ to be any set of pairs from $A \times B$, and a function $f : A \to B$ to be any relation where there is exactly one pair $(a, b) \in R$ for each $a \in A$. Standard properties and operations on functions and relations (reflexivity, transitivity, injection, bijection, composition) are defined in the usual way, and in particular composition is defined outside-in, so that $(f \circ g)(x) \overset{\text{def}}{=} f(g(x))$. A multiset over A is a function $S : A \to \mathbb{N}$.

We will occasionally use the term *mapping*, which in our work will be synonymous with *function* but which implies the use of some syntactic conveniences. For example, we may write $x \mapsto y$ to state that $(x, y) \in f$ for some contextually-appropriate f. In a similar fashion, $g\{x \mapsto y\}$ is an *update* to function g; it represents g except any $(x, a) \in g$ is removed and (x, y) is added.

3.1.2 Fast-Growing Functions

Somewhat recently a nomenclature has been adopted for sharing and reasoning about very large complexity classes. We shall make use of conventions as laid out by Czerwiński and Orlikowski in their proof of Ackermann-hardness for reachability [30] which are similar to the presentation by Schmitz [141]. The idea is to underpin a family of complexity classes by a family of corresponding fast-growing functions.

The family of functions will be written F, with $F_1, F_2, \ldots$ being the functions in the family. The definition is given recursively as follows:

$$F_1(n) = 2n$$
$$F_k(n) = \underbrace{F_{k-1} \circ \cdots \circ F_{k-1}}_{n \text{ times}}(1)$$

The definition of F coincides with Knuth's up-arrow notation [89], where

$$F_1(n) = 2n$$
$$F_2(n) = 2 \uparrow n$$
$$F_3(n) = 2 \uparrow\uparrow n$$

and so on. It may also be described as a specific instantiation of the family of hyperoperations (multiplication, exponentiation, tetration, pentation, et cetera).

It remains to formally define the corresponding family of complexity classes, which we shall write as $\mathcal{F} = \mathcal{F}_2, \mathcal{F}_3, \ldots$. We shall say that a problem with size n is a member of complexity class $\mathcal{F}_\alpha$ ($\alpha > 1$) if it is computable by a deterministic Turing machine in time $O(F_\alpha^c(n))$ for some constant c.

The complexity class TOWER, which was briefly the best lower bound of the reachability problem, is $\mathcal{F}_3$, corresponding to a tower of exponentials of height n. PR (the complexity class corresponding to the primitive recursive functions) is included in $\mathcal{F}$; more precisely, for any primitive recursive function f there is some k such that $f \in O(F_k)$.

We distinguish one special use of $\mathcal{F}$, called $\mathcal{F}_\omega$, where the corresponding function $F_\omega(n)$ is defined as $F_n(n)$. $\mathcal{F}_\omega$ is the complexity class ACKERMANN; any Ackermannian function grows faster than any fixed $\mathcal{F}_k$. This notation is slightly abused to produce variants on ACKERMANN such as cubic-Ackermann, written $\mathcal{F}_{\omega^3}$. The latter was briefly the best upper bound on the reachability problem; the problem is now known to be complete for ACKERMANN (per Section 2.2.2).

3.2 Vectors

Definition 3.1. A vector v in dimension d is a d-tuple $(v[1], v[2], \ldots, v[d])$.

Each position of a vector is called a *coordinate*. Note that we choose to start indexing vectors from 1—there is no "zeroth" coordinate. Indexing into a vector is performed with square brackets, e.g. in the vector $u = (4, 5, 6)$, the value of the third coordinate is $u[3] = 6$. Unless otherwise evident, values in vectors range over $\mathbb{N}$ and so a vector in dimension d is a member of $\mathbb{N}^d$.

Arithmetic operations, namely addition and subtraction, are extended to vectors pointwise, so that $(a, b, \ldots) + (x, y, \ldots)$ is $(a + x, b + y, \ldots)$. Scalar multiplication of vectors, of the form $\lambda(a, b, \ldots)$, is defined as $(\lambda \cdot a, \lambda \cdot b, \ldots)$. The concatenation of two vectors v and w is written $v \cdot w$. Vectors will never be multiplied by vectors.

We shall use a couple of notational shorthands for particular types of vectors. We may wish to refer to a vector composed all of zeroes, or all of ones, etc. In such cases we shall overline the value with a vector arrow: $\vec{0}$, $\vec{1}$, etc. The dimensionality of these overlined vectors is not fixed and should be inferred from context. Additionally we may occasionally abuse the notation of mappings for vectors; a vector denoted by $\{i \mapsto x\}$ is the zero vector with the ith coordinate set to x. The technical relationship between vectors and mappings is discussed in Section 4.5.

3.2.1 Well Quasi Orders and Pointwise Ordering

Many of the algorithms presented in this thesis derive their proof of termination from a notion of *well-quasi-ordering*. The Kleene closure A^* is the set of finite sequences of elements of A [88]. Let us extend this notion to include infinite sequences: we define A^ω to be the set of infinite sequences over A, and $A^\circledast = A^* \cup A^\omega$ all sequences over A.

Definition 3.2. A *quasi-ordering* $(\preccurlyeq, A)$ over a set A is a reflexive, transitive binary relation.

Definition 3.3. A *well-quasi-ordering* $(\leq, A)$ over A is a quasi-ordering over A such that in any infinite sequence $(a_1, a_2, \ldots) \in A^\omega$ there exist some $i, j \in \mathbb{N}_+$ such that $i < j$ and $a_i \leq a_j$.

Definition 3.4. The *pointwise ordering* $(\leq_\bullet, A)$ over a set A of vectors in dimension d is a relation such that $v \leq_\bullet w$ if and only if $v[i] \leq w[i]$ for all $i \in \{1, \ldots, d\}$.

Lemma 3.1 (Dickson's Lemma [38], as reformulated by Figueira et al. [49])**.** $(\leq_\bullet, \mathbb{N}^d)$ *is a well-quasi-ordering for all* $d \in \mathbb{N}$.

For the remainder of this work, unless specified otherwise, we shall take $\leq$ for vectors to be the pointwise ordering $\leq_\bullet$. Lemma 3.1 shall be relied upon extensively when discussing termination of procedures over systems of these vectors.

3.3 Well-Structured Transition Systems

Finkel and Schnoebelen give a generally applicable definition of well structured transition systems, which we shall recapitulate briefly with localised notation.

Definition 3.5 ([52, Section 2.2])**.** A *transition system* is any structure which includes

- A set S of *states*;
- A *transition relation* $\rightarrow \subset S \times S$.

Definition 3.6. [52, Section 2.3] A *well-structured transition system* is a transition system equipped with some relation $\leq \subseteq S \times S$ which is:

- a well-quasi-ordering; and
- *upward-compatible*: if $s_1 \rightarrow s_2$, then for any $t_1 \geq s_1$ there is some $t_2 \geq s_2$ such that (t_1, t_2) is in the reflexive transitive closure of $\rightarrow$.

3.4 Vector Addition Systems

We will now proceed to the definitions for the automata that form the basis of this work. To distinguish automata from vectors, automaton constructs will be written as some tuple in ⟨algebraic brackets⟩. The distinction between tuples and automaton constructs is nominal, and such objects can and will be treated like any other tuple.

Definition 3.7. A *Vector Addition System* (VAS) in dimension d is a finite set of transitions $W \subseteq \mathbb{Z}^d$.

The definition given above is distinct from the original formulation of Karp and Miller [87, Chapter IV] in that it omits the initial vector. When we formulate queries such as the reachability and coverability queries of Chapters 4 and 5, we shall include the initial vector as part of the query. This is for symmetry with the formulation of Petri nets and queries thereover.

A generalised version of VAS, originally defined by Hopcroft and Pansiot in 1979 [79], includes control states which limits when transitions are allowed to fire.

Definition 3.8. A *Vector Addition System with States* (VASS) in dimension d is a pair $V = \langle Q, T \rangle$, where

- Q is a finite set of states;
- $T \subseteq (Q \times \mathbb{N}^d \times Q)$ is a finite set of transitions.

It is this formulation that we will make use of in the remainder of the work. As noted by Hopcroft and Pansiot, the addition of states often reduces the dimension needed to model a particular system and makes the relationship between vectors and states more apparent. Observe that any VAS W can be viewed as a VASS $\langle \{\circ\}, \{(\circ, w, \circ) \mid w \in W\} \rangle$.

3.4.1 Configurations and Runs

Let us define the semantics of Vector Addition Systems (with states).

The critical piece of VASS semantics, which distinguishes them from arbitary state machines, is this: During operation, the machine maintains a configuration, which includes a vector. At no point during the operation of a VASS is any value in the vector of its configuration allowed to become negative.

Definition 3.9. A *configuration* of a VASS $\langle Q, T \rangle$ in dimension d is a pair (q, v) for some $q \in Q$ and $v \in \mathbb{Z}^d$. Such a configuration is *valid* if and only if $v \in \mathbb{N}^d$.

Definition 3.10. In a VASS $V = \langle Q, T \rangle$, transition $t = (q, w, r)$ is *active* (equivalently *can be fired* or *is fireable*) from some configuration (q', v) in V if and only if $q = q'$ and $(r, v + w)$ is a valid configuration. If so, we may write

$$(q, v) \xrightarrow{t} (r, v + w).$$

Operation of a VASS consists of firing transitions which lead to valid configurations. We call such sequences of firings *runs*.

Definition 3.11. A *run* on a VASS $V = \langle Q, T \rangle$ is a sequence of transitions $(t_1, \cdots, t_\ell)$ from T such that $(q_0, v_0) \xrightarrow{t_1} (q_1, v_1) \xrightarrow{t_2} \cdots \xrightarrow{t_\ell} (q_\ell, v_\ell)$. The length of such a run is ℓ. (q_0, v_0) and (q_ℓ, v_ℓ) are called the *initial* and *final* configurations of the run respectively.

Queries on VASSs made in this work will consist of asking whether a run exists which satisfies certain (equality or inequality) constraints on its initial and final configurations. In Chapter 5, and in particular Section 44, we shall also interest ourselves in *pseudo-runs*: runs where the intermediate configurations $(q_1, v_1), \ldots, (q_{\ell-1}, v_{\ell-1})$ are not required to be valid. The semantics of configurations and runs of VAS are derived via the semantics of a VASS with a single state.

It is often convenient to consider the properties of a run notwithstanding the precise order of transition. In such cases, we can choose a more succinct representation of runs. This succinct representation is typically referred to as the Parikh image, after an analogous technique devised by Parikh for use in the theory of context-free languages [122].

Definition 3.12. For a run $\sigma = (t_1, \ldots, t_\ell)$ on a VASS $V = \langle Q, T \rangle$, the Parikh image $\pi_\sigma : T \to \mathbb{N}$ is a count of the number of occurrences of each transition of T in σ. Formally:

$$\pi_\sigma(t) = |\{i \mid i \in \{1, \ldots, \ell\}, t_i = t\}|$$

The Parikh image will be of particular value when computing relaxations of runs in Chapter 5.

3.4.2 VASS Diagrams

We will draw VASS in a way similar to the standard presentation of state-transition diagrams for finite automata. We render a VASS as a directed multigraph, where each state $q \in Q$ is denoted as a node labelled by q and each transition (q, v, r) is denoted as an edge from q to r labelled by v. Unlike finite automata, initial and final states do not form part of the model (hence do not appear in the diagram)—instead they are included as part of any queries asked of it.

Example 3.1. Consider the VASS $V_{\mathsf{ex}} = \langle Q, T\rangle$ in dimension 2, where

$$Q = \{q, r\}$$
$$T = \{(q, (-1, 1), q), (q, (0, 1), r)\}$$

The diagram corresponding to V_{ex} is given in Figure 3.1. (We shall make use of V_{ex} again in Chapter 5.) From the diagram it is evident that, once state r is reached, it is no longer possible to fire any transitions. Hence we can infer that transition t_1 may be fired at most once in any run on V_{ex}.

Each of the following is a run on V_{ex}:

- $(r, (0, 0))$;
- $(q, (1, 0)) \xrightarrow{t_1} (r, (0, 0))$;
- $(q, (2, 2)) \xrightarrow{t_0} (q, (1, 3)) \xrightarrow{t_0} (q, (0, 4))$.

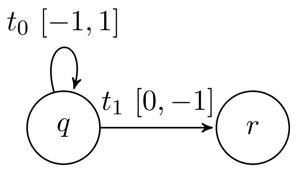


Figure 3.1: An example VASS, V_{ex}.

3.5 Petri nets

A Petri net is an alternative formulation of a very similar idea. Instead of using states and vectors to move between configurations as in VASS, instead we mark *places* with *tokens* and use transitions to shuffle tokens between these places.

Definition 3.13. A Petri net is a triple $\mathcal{N} = \langle P, T, f\rangle$, where:

- P is a finite set of *places*;
- T is a finite set of *transitions*;
- $f : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the *flow function.*

Both P and T are arbitrary in the sense that we can use arbitrary labels for the places and transitions of a Petri net. In this work we endeavour to stick with $p_1, \ldots, p_n$ and $t_1, \ldots, t_n$ for the places and transitions of a net respectively. As a notational convenience, we may write $(a, b) \mapsto n$ when $f(a, b) = n$; n is called the *flow* from a to b.

3.5.1 Behaviour

Much like VASS configurations, in Petri nets we define a notion of the current configuration of a Petri net. Configurations of Petri nets are called *markings*. A marking is characterised by the placement of so-called *tokens*. Such tokens are fungible and so it is sufficient to count them:

Definition 3.14. A *marking* of a Petri net $\mathcal{N} = \langle P, T, f \rangle$ is a mapping $M : P \to \mathbb{N}$.

In words, a marking associates to each place the number of tokens that are currently hosted in that place. We shall use the same pointwise notion of ordering for markings as for vectors: $M \leq M'$ if and only if $M(p) \leq M'(p)$ for all $p \in P$. The number of tokens in places will increase and decrease over time as the result of firing transitions:

Definition 3.15. For a Petri net $\mathcal{N} = \langle P, T, f \rangle$, a transition $t \in T$ is *enabled* (equivalently *can be fired*) in a marking M if and only if, for every place $p \in P$, we have $M(p) \geq f(p, t)$. The result of firing such a transition is $M' = \{(p, M(p) - f(p, t) + f(t, p)) \mid p \in P\}$, and we may write $M \xrightarrow{t} M'$.

Runs are defined analogously to VASS:

Definition 3.16. A *run* on a Petri net $\mathcal{N} = \langle P, T, f \rangle$ is a sequence of transitions $t_1, \cdots, t_\ell$ of T such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_\ell} M_\ell$. The length of such a run is ℓ. M_0 and M_ℓ are the *initial* and *final* markings of the run respectively.

As with VASS, the ability to fire a transition is predicated on having sufficient values in some set of locations as determined by the transition. From time to time it will be convenient to refer to that set of locations (and some similarly-defined sets) by name.

Definition 3.17. The *pre-set* $\mathbf{pre}(t)$ of a transition t in net $\mathcal{N} = \langle P, T, f \rangle$ is the set of places $p \in P$ from which tokens are removed when t is fired. The *post-set* $\mathbf{post}(t)$ of t is the set of places into which tokens are added when t is fired. Formally:

$$\mathbf{pre}(t) = \{p \in P \mid f(p, t) > 0\}$$
$$\mathbf{post}(t) = \{p \in P \mid f(t, p) > 0\}$$

Note that a place may appear both in the pre-set and post-set of a transition. As is common, we shall overload the notation to extend the pre-set and post-set to places as well:

Definition 3.18. The *pre-set* $\mathbf{pre}(p)$ of a place p in a net $\mathcal{N} = \langle P, T, F \rangle$ is the set of transitions $t \in T$ that add tokens to p when t is fired. The *post-set* $\mathbf{post}(p)$ of p is the set of transitions that remove tokens from p when t is fired. Formally:

$$\mathbf{pre}(p) = \{t \in T \mid f(t,p) > 0\}$$
$$\mathbf{post}(p) = \{t \in T \mid f(p,t) > 0\}$$

3.5.2 Petri net Diagrams

It is in diagramming Petri nets that the value of the formalism starts to become apparent. We draw a Petri net $\mathcal{N} = \langle P, T, f \rangle$ as a directed bipartite graph. Each place $p \in P$ is rendered as a grey circle, and each transition t is rendered as a black rectangle (the orientation of the rectangle is arbitrary). The directed edges (often called *arcs*) between places and transitions are drawn according to the flow relation. A flow $(a, b) \mapsto n$ with value $n > 1$ is labelled with n; edges without a label are assumed to have flow 1. Edges with flow zero are not drawn.

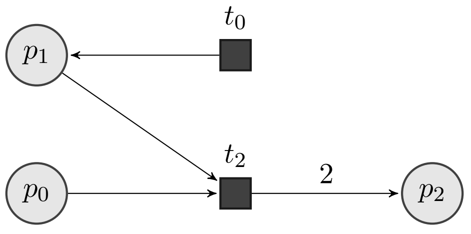
Example 3.2. Consider the Petri net $\mathcal{N}_{\mathsf{ex}} = \langle \{p_0, p_1, p_2\}, \{t_0, t_1, t_1\}, f \rangle$, where the nonzero flows in f are

$$(t_0, p_0) \mapsto 1 \quad (p_0, t_1) \mapsto 1 \quad (p_1, t_2) \mapsto 1 \quad (t_1, p_2) \mapsto 2.$$

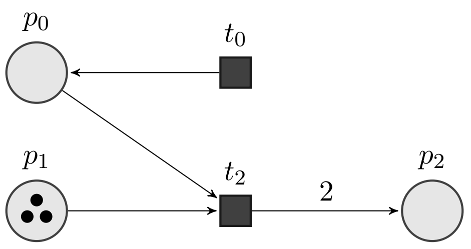
Then we may diagram $\mathcal{N}_{\mathsf{ex}}$ as in Figure 3.2a. If we wish to diagram a Petri net with some particular marking M, we put on the circle corresponding to each place p a number of black discs equal to $M(p)$. Figure 3.2b shows $\mathcal{N}_{\mathsf{ex}}$ with marking $\{p_1 \mapsto 3\}$ and all other places 0. To distinguish an unmarked net from a net with a marking with all places zero, in an unmarked net we write the label for each place inside the circle for that place.

3.6 Petri nets versus Vector Addition Systems

One may infer from the definitions given above that Petri nets and Vector Addition Systems with States are quite similar. Both formalisms revolve around a notion of transitions, which manipulate values in a number of locations. Both require that in order for firing those transitions to be allowed, the firing may not result in values in any of those locations becoming negative. And both



(a) $\mathcal{N}_{\mathsf{ex}}$, unmarked.



(b) $\mathcal{N}_{\mathsf{ex}}$ with marking $\{p_1 \mapsto 3\}$ as denoted by tokens.

Figure 3.2: The net $\mathcal{N}_{\mathsf{ex}}$ as defined in Example 3.2, rendered without and with a marking.

have a shared notion of runs, which are sequences of such firings. However, there are a couple of properties of each that make translation to the other not immediate. It was quickly established that Petri nets and Vector Addition Systems are equivalent formalisms—that is, they are equally expressive—and this was formally shown by Hack in 1974 [75]. We will present a family of translations between Petri nets and VAS, and in doing so show also that Vector Addition Systems (the ones without states) are as expressive still.

Definition 3.19. Let us say that X *(weakly) simulates* Y if we can define an injective function R from the configurations of Y to the configurations of X such that there exists a run from a to a' in Y if and only if there exists a run from $R(a)$ to $R(a')$ in X.

Definition 3.19 is designed to abstract over VASs, VASSs, and Petri nets. We will show equivalence of the three models by giving a triplet of one-way simulations. Observe that the simulation property is transitive and so we get that each of the three models can simulate the others.

$$\begin{tikzcd} & \text{Petri nets} \arrow[dr, two heads, "\text{Thm. 3.3}"] & \\ \mathsf{VAS}\text{s} \arrow[ur, two heads, "\text{Thm. 3.2}"] & & \mathsf{VASS}\text{s} \arrow[ll, two heads, "\text{Thm. 3.1}"] \end{tikzcd}$$

Figure 3.3: The triplet of one-way simulations given below, where $Y \twoheadrightarrow X$ means X simulates Y.

Theorem 3.1 ([79, Lemma 2.1])**.** *For every* VASS $V = \langle Q, T\rangle$ *in dimension* d*, there is a* VAS W *in dimension* $d + 3$ *that simulates it.*

Proof (sketch). The state of any given configuration of V is encoded directly into the three additional coordinates of W. The simulation property is maintained by a clever bookkeeping method.

Without loss of generality, assign some numbering $q_1, \dots, q_k$ to the k states of V. For each $i \in \{1 \dots k\}$, define the intermediate value $b_i = (k+1)(k+1-i)$. We may then define R as

$$R(q_i, v) = v \cdot (i, b_i, 0).$$

Transitions in V are translated to W in the following way. In each of the following rules, every possible instantiation of the antecedent (above the line) induces the corresponding consequent (below the line). This notation shall be reused from Chapter 6 onwards.

$$\frac{i \in \{1, \dots, k\}}{\vec{0} \cdot (-i, k + 1 - i - b_i, b_{k+1-i}) \in W} \quad (1)$$

$$\frac{i \in \{1, \dots, k\}}{\vec{0} \cdot (b_i, i - (k + 1), i - b_{k+1-i}) \in W} \quad (2)$$

$$\frac{(q_i, v, q_j) \in T}{v \cdot (j - b_i, b_j, -i) \in W} \quad (3)$$

The intuition behind the encoding is as follows. Suppose we wish to apply some transition $q_i \xrightarrow{w} q_j$. If starting in state q_i the system shall have configuration $R(q_i, v)$ for some v. Reaching $R(q_j, v + w)$ is achieved by first applying transitions (1) and (2) for i, leaving the last three coordinates in position $(b_i, 0, i)$. Observe that, thanks to the value of b_i being greater than a_j for any i, j, it will always be possible to apply those transitions in the desired order. It remains to apply transition (3), which will leave the system at

$(v + w) \cdot (j, b_j, 0) = R(q_j, v + w)$, as desired. If i is too great or too small then transition (3) will not be enabled as either the first or third position would become negative, and so it is only enabled after firing both (1) and (2) for that value of i. Hence this scheme does not allow for firings which would result in visiting erroneous states. $\square$

Theorem 3.2. *For every* VAS *W in dimension d there is a Petri net $\mathcal{N} = \langle P, T, f\rangle$ that simulates it.*

Proof. This is the simplest of the three constructions. Let $P = \{1, \dots, d\}$, $T = W$, and the nonzero flows of f (written f_+) be defined as follows.

$$\begin{aligned} f_+ &= f_{pt} \cup f_{tp} \\ f_{pt} &= \bigcup_{t \in T} \{(p, t) \mapsto -t[p] \mid t[p] < 0\} \\ f_{tp} &= \bigcup_{t \in T} \{(t, p) \mapsto t[p] \mid t[p] > 0\} \end{aligned}$$

The simulating function R, which identifies a marking of $\mathcal{N}$ for each configuration of W, is

$$R(v) = \{i \mapsto v[i] \mid i \in \{1, \dots, d\}\}.$$

R is clearly bijective. Moreover, the encoding of transitions from W as transitions where (p, t) and (t, p) are not both nonzero is also bijective. Observe that the firing conditions of the transitions in the net are analagous to the firing conditions of the vectors in the VAS. It follows that $\mathcal{N}$ simulates W. $\square$

We finish the virtuous triangle by simulating Petri nets in VASS.

Theorem 3.3. *For every Petri net $\mathcal{N} = \langle P, T, f\rangle$, there is a* VASS *$V = \langle Q_V, T_V\rangle$ in dimension $d = |P|$ that simulates it.*

Proof. Without loss of generality, let us arbitrarily number the places of the net $\{p_1, \dots, p_d\}$. Markings will be represented as vectors in dimension d.

The minor hurdle to overcome here is that in Petri nets we separate out the flow into and out of a transition—some transition may remove five counters to add six back. In VASS we have no such luxury in a single transition, but we can simulate this behaviour with two consecutive transitions.

For each transition t, we define

$$\begin{aligned} v_{\mathbf{pre}}(t) &= (f(p_0, t), f(p_1, t), \cdots, f(p_n, t)) \\ v_{\mathbf{post}}(t) &= (f(t, p_0), f(t, p_1), \cdots, f(t, p_n)) \end{aligned}$$

Intuitively these vectors correspond to the "in-flow" and "out-flow" for each transition. Observe that all the values in every coordinate in every vector $v_{\mathbf{pre}}(t)$ or $v_{\mathbf{post}}(t)$ will be nonnegative. It remains to concatenate transitions corresponding to these vectors in such a way that we can only fire the post-vector immediately after the pre-vector. Q_V and T_V are as given below.

$$Q_V = \{\mathrm{mid}_t \mid t \in T\} \cup \{\bullet\}$$

$$\begin{aligned} T_V &= T_{pre} \cup T_{post} \\ T_{pre} &= \{(\bullet, -v_{\mathbf{pre}}(t), \mathrm{mid}_t) \mid t \in T\} \\ T_{post} &= \{(\mathrm{mid}_t, v_{\mathbf{post}}(t), \bullet) \mid t \in T\} \end{aligned}$$

The state $\bullet$ is a single distinguished state where configurations of V in that state are exactly the markings of $\mathcal{N}$. The additional states mid_t are used for bookkeeping, where the removal part of the Petri net transition occurs first with vector $-v_{\mathbf{pre}}(t)$ and from there the only transition back to $\bullet$ is via the transition with vector $v_{\mathbf{post}}(t)$, which can always be fired. The simulating function R is

$$R(M) = (\bullet, (M(p_1), M(p_2), \ldots, M(p_d))).$$

□

It is worth briefly remarking on the algorithmic complexity of these constructions. Each of the constructions results in a machine which has the same dimension (in Petri nets, number of places), with the exception of the translation from VASS to VAS which increases the dimension from d to $d + 3$. In each case the number of transitions in the simulating machine is linear in the number of transitions in the simulated machine, and the constructions can all be performed in polynomial time and with logarithmic space.

The upshot of the above results is that Petri nets and Vector Addition Systems (with or without States) can be considered equivalent formalisms, and any question posed over one of the models has an analogous question that can be posed over the others. We shall use this equivalence liberally in the chapters to come. With this in mind, we are now ready to embark on the theoretical contributions of this work.

Chapter 4

The Coverability Problem & HCover

Let us begin with an exploration of coverability.

As very recently covered in Section 3.6, it is known that Petri nets and Vector Addition Systems (with or without states) are equivalent formalisms, such that problems defined over one may be reformulated as problems over the other. It is commonplace in the literature for theoretical results to be presented over VAS or VASS, whereas practical applications of the formalisms will often make use of Petri nets. Sections 3.5 and 3.4 offer examples of the graphical representation of each of the models.

The coverability problem can easily be formulated over either model. We present both formulations here.

Definition 4.1. The *coverability problem for Petri Nets* asks, given a Petri net $\mathcal{N} = \langle P, T, \mathcal{F} \rangle$, an initial marking M_0, and a target marking M, whether there is any run $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M'$ such that $M' \geq M$.

Definition 4.2. The *coverability problem for Vector Addition Systems with States* asks, given a VASS $V = \langle Q, T \rangle$, an initial configuration (q_0, v_0), and a target configuration (q, v), whether there exists a run

$$(q_0, v_0) \xrightarrow{t_0} (q_1, v_1) \xrightarrow{t_1} \cdots \xrightarrow{t_n} (q, v')$$

such that $v' \geq v$.

The analogy between the two formulations should be obvious, and translation of problems from one to the other is unremarkable. Recall that both VASS configurations and Petri net markings make use of the pointwise notion of ordering defined in Section 3.2.1.

The reason for defining both forms is to justify the use of the theoreticallyminded Vector Addition Systems formulation of the coverability problem (as we shall see later in this chapter) to work with instances and examples given in the more readily diagrammed format of Petri nets.

4.1 Why Coverability?

As remarked on in Chapter 2, Petri nets have applicability across a wide range of problem domains, and the coverability problem is able to capture a large number of properties of interest.

Coverability is intrinsically tied to a notion of safety. A Petri net is considered *safe* with respect to some predicate f and initial marking M_0 if and only if, for every marking M' reachable from M_0, $f(M')$ holds. Safety predicates are formulated as bounds on individual places, for example "Place p must have at most 1 token". Such a safety property can be tested by checking for (non)coverability of the marking $\{p \mapsto 2\}$. Such safety properties are sufficient for many forms of model checking, both in the theory of hardware and software (checking for critical section reuse, for example) and further beyond in business process modelling, resource allocation, or safety-critical systems such as transportation control infrastructure.

Coverability is also significantly less computationally intensive to check than reachability, being (worst-case) double-exponential in running time compared to the Ackermannian running time of procedures for reachability. Hence there is an appetite to reduce problems to coverability instances rather than reachability wherever possible.

We shall now begin on the technical details of solving the coverability problem.

4.2 Upward-closure

Many algorithms for solving coverability rely on a notion of *upward-closure*:

Definition 4.3. A set of vectors $V \subseteq \mathbb{N}^d$ is *upward-closed* if and only if, for all $v \in V$ and $v' \in \mathbb{N}^d$, if $v' \geq v$, then $v' \in V$. The *upward-closure* $\uparrow v$ of a vector v is the (unique) smallest upward-closed set containing v; the upward-closure of a set of vectors $\uparrow V$ is the union of the upward-closures of the members of V.

We define downward closure ($\downarrow$) identically but with $\leq$ rather than $\geq$. With these definitions in mind, we may reformulate coverability as finding some run ending at a vector v' contained in $\uparrow v$ (or equivalently where v is contained in $\downarrow v'$).

All upward-closures are infinite, and so it is not possible to immediately use them in coverability procedures. Instead, we require a way of representing such upward-closed sets in a finite (preferably small) form. We elect to represent an upward-closed set in terms of some finite subset of its members.

Dickson's lemma [38] states that for any such set S, there is some minimal subset of representative elements—and that subset is both finite and unique. We call that subset the *basis* for S. The calculation of a basis for an upward-closed set, given a superset of that basis, is a significant computational bottleneck in practice. A naive approach would require $O(n^2d)$ time, with n the size of the subset and d the dimension of the vector space. Among other things, we seek to address this bottleneck as part of the work undertaken in this chapter.

4.3 Approaches to Coverability

A number of approaches have been described for solving coverability since the problem was formulated by Karp and Miller in 1969 [87]. Here we shall stick to presenting the theoretical results via Vector Addition Systems without states.

4.3.1 The Karp-Miller Coverability Procedure

Let us define the *coverability set* $\downarrow V, (q, v)$ of the VASS $V = \langle Q, T \rangle$ from configuration (q, v) as the set of all configurations that are coverable in V by some run starting at (q, v). Asking whether (q', v') is coverable from (q, v) in V is therefore the same as testing for membership of (q', v') in $\downarrow V, (q, v)$.

For this membership test to serve as a meaningful way of deciding coverability, we would need some algorithm for generating the complete coverability set in finite time. Fortunately, Karp and Miller [87] have provided us just such an algorithm.

More precisely, the algorithm will generate the so-called *Karp-Miller tree* for a given V and starting configuration (q, v). The Karp-Miller tree is a tree where every node is labelled by an *ω-configuration* and every edge is labelled by a transition from T.

Definition 4.4. For some VASS $V = \langle Q, T \rangle$ in dimension d, an *ω-configuration* is a pair $(q, v_\omega) \in Q \times (\mathbb{N} \cup \{\omega\})^d$.

Intuitively, this gives us the ability to replace any coordinates in a configuration by ω. Every configuration of a VASS is also an ω-configuration. When an ω appears for some coordinate at a node in the Karp-Miller tree, that indicates that that coordinate can be pumped arbitrarily high by repeatedly firing the transitions labelling some (contiguous) subsequence of the path from the root to that node. The set of all labels in the tree is a representation of the coverability set. We shall see shortly that this representation is finite.

Algorithm 1 Karp-Miller Tree Computation

```
1: procedure KM-Tree($V = \langle Q, T \rangle, (q, v), A$ (initially $\emptyset$))
2:     if $\exists\ (q, v_a) \in A$ s.t. $v_a \geq v$ then
3:         return $\emptyset$                                    ▷ A sink; end exploration.
4:     end if
5:     for $(q, v_a) \in A, i \in \{1, \ldots, d\}$ do
6:         if $v_a < v \wedge v_a[i] < v[i]$ then
7:             $v[i] \leftarrow \omega$                           ▷ Mark pumped places by $\omega$.
8:         end if
9:     end for
10:    $D \leftarrow \bigcup\{\text{KM-Tree}(V, (q', v'), A \cup \{(q, v)\}) \mid \exists t \in T : (q, v) \xrightarrow{t} (q', v')\}$
11:    return $\{((q, v), D)\}$
12: end procedure
```

In the version of the Karp-Miller tree construction presented in Algorithm 1, we denote a node as a pair (c, D) where c is some ω-configuration labelling the node and D is the set of child nodes. Labels on nodes need not be unique in the whole tree, but they will be unique on any path to the root. In short, we exhaustively follow every legal path from the initial configuration (q, v) in V.

At each point, if our node (q, v) is pointwise smaller than or equal to some ancestor (q, v_a) with the same state, there are no longer any interesting configurations to find on this path and we stop expanding it. If our vector is strictly *larger* than it was at some ancestor with the same state (taking ω to be larger than any natural), then we set any coordinates which are strictly larger to ω and proceed. We compute the set of configurations reachable in one step from this node and recursively build the tree. In building the set of configurations reachable in one step, we permit any increase or decrease of an ω coordinate; that coordinate remains ω.

Consider any path starting at the root. Any time we revisit a state on the path, either (1) we stop expanding the path; (2) we set at least one coordinate to ω; or (3) the new and old values are incomparable under pointwise ordering. Case (2) can only happen at most d times on any path (assuming vectors in dimension d); by Dickson's lemma the number of occurrences of (3) is finite. Hence any path must be finite. Since the branching at any node is bounded by the (finite) number of transitions, by König's lemma the tree is finite and this procedure will terminate.

The Karp-Miller tree is used in a number of other applications. For example, this algorithm shall be used in Chapter 5 as a subroutine in KReach, our implementation of Kosaraju's algorithm for reachability in VASS. The algorithm has since been extended to post-self-modifying nets [143], ω-recursive nets [53], and branching VASS [147]. A work of Blondin, Finkel and Goubalt-Larrecq [14] provides a holistic appraisal of the Karp-Miller coverability procedure and its applications to many flavours of well-structured transition systems.

4.3.2 Search methods

There are two varieties of search algorithm which can decide coverability.

1. *Forward search.* From the initial vector v_0, all vectors in $\downarrow v_0$ can trivially be covered. Forward methods start with $\downarrow v_0$ and repeatedly fire all possible transitions, computing a growing downward-closed set of coverable vectors, until that set contains the target vector v, or there are no remaining transitions whose firing grows the set.

2. *Backward search.* These methods start with the upward closure of the final vector $\uparrow v$ and add additive vectors "in reverse", until v_0 is contained in the set, or there are no remaining ways to grows the set. (We shall discuss what adding vectors "in reverse" means shortly.)

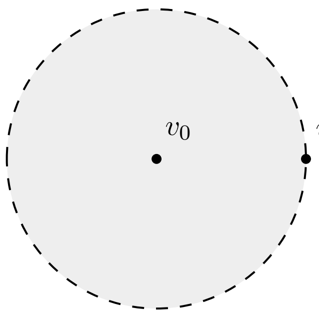
The two search methods described above are symmetrical in some sense, but each one admits a different set of optimisations. Both forward and backward search are guaranteed to terminate [52, 87]. Section 4.3.1 outlines a classic forward search method.

Other methods have been trialled, such as in the `MIST` coverability checking tool [59], which combine these two mechanisms: running both forward and backward algorithms at the same time in an attempt to reduce the total space explored. Figure 4.1 gives a (low-dimension) geometric exposition of the logic behind this approach; the computational saving increases exponentially as the dimensionality of the space increases.

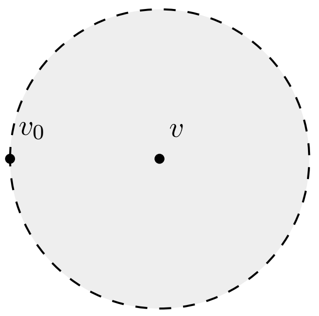
However, this dual search mechanism is not ideal for all problem domains. For example, in cases where the prevalence of coverable instances is very low, performing both searches may be less efficient, as the entire forward and backward search space may need to be explored before coverability can be ruled out. In addition, checking for overlap between the sets generated by forward and backward coverability is itself computationally intensive and introduces its own significant overhead.

4.3.3 Augmented search methods

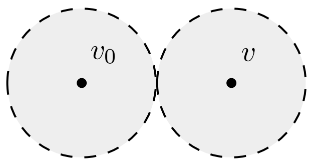
More recent approaches to deciding coverability commonly rely on solving *relaxations* of the coverability problem. One method of particular interest relaxes the rules of runs over the vector addition system.



(a) Forward search starts with $\downarrow v_0$ and ends when the downward-closed set contains v or cannot be grown.



(b) Backward search starts with $\uparrow v$ and ends when the upward-closed set contains v_0 or cannot be further expanded.



(c) By performing forward and backward search at the same time, the overall "radius" of the search is smaller and less irrelevant search space is explored overall.

Figure 4.1: A geometric representation of the search space for forward and backward coverability search.

In [13], Blondin et al. make use of reachability and coverability in the continuous setting, where one may add arbitrary fractions of vectors during runs. The continuous setting admits some efficient approaches—indeed, Fraca and Haddad [56] showed that not only $\mathbb{Q}$-coverability, but $\mathbb{Q}$-reachability, is computable in polynomial time. This relaxation can quickly and effectively prune the search space, as laid out in Section 28 "Continuous VASS".

The technique of pruned search was expanded on by Geffroy, Leroux and Soutre [63] to include more methods of pruning. Their ideas involve computing inductive invariants of the net that statically preclude sets of vectors fufilling certain criteria (see Section 30 "Inductive Invariants") from ever bring part of a covering run. This further reduces unneeded computation.

4.3.4 Coverability by Invariant-driven Pruned Search

Invariant-driven pruned search, in essence the `ICover` algorithm of Geffroy, Leroux and Sutre described in [63], is a backward search exactly of the form described in Section 4.3.2. We reformulate their description in terms of VASS for the sake of simplicity.

The backwards search algorithm, presented in Algorithm 2, proceeds as follows. We begin with a VASS $V = \langle Q, T \rangle$, an initial configuration (q_0, v_0) and a target configuration (q, v). The intuition behind the backward coverability algorithm is that we inductively compute the set of all vectors which can, by some run, end up covering our target vector. We start with the upward closure of our target vector—everything in $\uparrow v$ trivially covers v, since by definition all elements of $\uparrow v$ are at least v with respect to our ordering $\leq$.

Recalling that we represent upward-closed sets by their minimal basis, we then, for each element u of that basis U, compute the maximal "pre-set" of vectors $W = \mathit{maxpre}(u)$ which, by some transition, would result in a vector in $\uparrow u$. W is also clearly upward-closed (if $w \xrightarrow{t} u$ for some $t \in V$, then for any $w' \geq w$, $w' \xrightarrow{t} u'$ such that $u' - u = w' - w \geq \vec{0}$ and therefore $u' \geq u$). Hence W has a computable, minimal, finite basis. Taking the union of the pre-set bases for all such elements u, along with U itself, gives us a new upward-closed set U'

Since all elements of U are in U', either U' strictly contains U or it is exactly equal to U. It is easy enough to detect either case—if the sets are the same then their bases are identical. If U' stritly contains U (we say it has "grown") then either U' contains our initial vector, in which case the target vector is known to be coverable from the initial vector and we finish with YES; or we can keep searching and we repeat the process described above. If U' has *not* grown, then we now have the full, upward-closed set of vectors which can cover v.

Algorithm 2 Simple backward coverability for VAS

```
 1: procedure Back-Cover($V, v_0, v$)                ▷ Is $v$ coverable from $v_0$ in $V$?
 2:     $U_0 \leftarrow \emptyset$
 3:     $U_1 \leftarrow \uparrow v$
 4:     $i \leftarrow 1$
 5:     while $U_{i-1} \subset U_i$ do
 6:         $i \leftarrow i + 1$
 7:         $U_i \leftarrow U_{i-1} \cup \uparrow \bigcup \{\mathit{maxpre}(u) \mid u \in U_i\}$
 8:         if $v_0 \in U_i$ then
 9:             return YES                              ▷ $v_0$ has been covered
10:         end if
11:     end while
12:     return NO                            ▷ The search space is exhausted
13: end procedure
```

Vectors with nonnegative components form a *well-quasi-ordering* under our definition of $\leq$ [38]. This means that any infinite sequence of distinct vectors $v_0, v_1, v_2, \cdots$ from $\mathbb{N}^d$ must contain some pair v_i, v_j such that $i \leq j$ and $v_i \leq v_j$. In our setting, we may equivalently say that there is no infinite sequence of distinct vectors v_i such that $v_0 \geq v_1 \geq v_2 \geq \cdots$. As a corollary, when considering upward-closed sets U_i of such vectors, any sequence $U_0 \subseteq U_1 \subseteq U_2 \subseteq \cdots$ will always stop growing eventually [52, Lemma 2.4]. The upshot of this is that we can only ever run a finite number of iterations before U can no longer grow, and so the procedure will always terminate. (Indeed, the well-quasi-ordering of vectors is the original derivation of this algorithm, which was defined generically over all well-structured transition systems.) Invariants are used to reduce the size of successive U terms and thus cause the procedure to converge and terminate more quickly. We shall now describe some such invariants.

Continuous VASS

The first time the continuous treatment of VASS was used as a pruning criterion for search was in [13] by Blondin et al. In the continuous setting, one is permitted to add non-negative rational amounts of any of the system's vectors to the current vector, with all the other constraints of VASS (non-negativity in runs in particular) respected.

Definition 4.5. A *continuous run* on a VASS $V = \langle Q, T \rangle$, starting at an initial configuration (q_0, v_0), is a sequence

$$(q_0, v_0) \xrightarrow{x_0(t_0)} (q_1, v_1) \xrightarrow{x_1(t_1)} \cdots \xrightarrow{x_n(t_n)} (q, v),$$

such that $\forall i: \ v_{i+1} = v_i + x_i \cdot t_i; \ v_i \geq \vec{0}; \ x_i \in \mathbb{Q}^+;$ and $t_i \in V$. Coverability where runs are replaced by continuous runs is called $\mathbb{Q}$-*coverability.*

$\mathbb{Q}$-coverability is a relaxation of coverability: any run in VASS can be treated as a continuous run where the coefficients are all 1. Therefore we can use (non-)coverability in the continuous setting as a pruning criterion for the standard setting. If no run exists with *any* suitable coefficients, then certainly no run exists with coefficients 1.

As laid out by Fraca and Haddad [56], it is possible to solve not only $\mathbb{Q}$-coverability but also the analogous $\mathbb{Q}$-*reachability* in polynomial time. It is their $\mathbb{Q}$-reachability algorithm which is used here to empower the pruning mechanism. The procedure, the details of which are the subject of [56, Algorithm 2], encodes the reachability properties into existential first-order logic and solves the resulting system via integer linear programming—a technique which shall reappear several times through Chapters 4 and 5. Constructing a ($\mathbb{Q}$-)reachability problem from a ($\mathbb{Q}$-)coverability problem is not challenging: we can augment any VASS with transitions such that, for a given v, v is reachable from any vector in $\uparrow v$, and so deciding reachability on the augmented VASS becomes equivalent to deciding coverability on the original. The construction for this is given in Section 5.1. Moreover the augmentation is small and so the algorithm is still polynomial.

The experimental results of [13] suggest that this alone is a significant improvement over many of the then state-of-the-art coverability checkers. Their tool `QCover` outperformed most contemporary solvers on uncoverable instances; only the `BFC` tool [85] was significantly faster on coverable instances.

Inductive Invariants

The techniques of Geffroy, Leroux and Sutre in [63] are broadly similar. As with Blondin et al. in [13], the algorithm proceeds by standard backward coverability and computation of growing upward-closed sets. The difference is in the pruning criteria.

Whereas Blondin et al. use exclusively $\mathbb{Q}$-coverability as their pruning criterion, Geffroy, Leroux and Sutre use a number of different efficiently-computable invariants to minimise the size of the basis set.

Observe that the subtraction of a downward-closed set from an upward-closed set is itself upward-closed. The approach of [63] is to identify as many downward-closed invariants as possible (for example, that no vector in $\downarrow v_1$ could possibly be reachable from v_0); and then subtract these downward-closed sets from the growing set U at each step. This reduces the number of steps before the series U_k of growing sets stabilises.

As with [13], the invariants are expressed efficiently using integer linear programming, in this case via a state equation. The state equation can be viewed as an encoding of the reachability conditions as an integer linear program, with the nonnegativity constraint on vector components removed: it measures only the displacement of each component due to some set of fired transitions, and ensures that it matches the difference between the initial and target vectors in that component. Nor does the state equation care about the order of firing: with the nonnegativity constraint removed, transitions may be fired in any order. To solve the state equation is to solve a relaxation of standard reachability. By reformulating the state equation with inequalities instead of equalities, we get the state *in*equation, which is the analogous system for coverability.

The state inequation itself is a downward-closed invariant: it provides an overapproximation of (the upward-closed set of) vectors from which a target vector may be coverable. The complement of that set is therefore downward-closed and represents the set of vectors which can never cover the target vector.

In their tool `ICover`, Geffroy, Leroux and Sutre removed the $\mathbb{Q}$-coverability based pruning of Blondin et al.'s `QCover` and replaced it wholesale with pruning based on their inductive invariants. In experiments, the two tools were able to decide almost exactly the same number of instances, though `ICover` solved them in 80% of the time of QCover. This is a positive indication that the techniques of ICover, which are inherently extensible, are at least as competitive as QCover and may present a useful starting point for the exploration of further invariants for pruning. It is for this reason that the ICover-based pruning algorithm was used as a basis for the tool presented in this chapter.

4.4 HCover

The coverability procedure outlined above, invariant-driven pruning, includes the use of state-space search via backward coverability; static computation of downward-closed invariants; and integer linear programming. Taken together, the algorithm is something of a *grand tour* of the highlights of decision procedure techniques for Vector Addition Systems with States. It therefore makes an excellent introductory topic for a newcomer to this field of study. In this section we introduce and discuss the implementation of HCover, an implementation of the ICover algorithm, presented in a performant, digestible, and extensible way.

As one may anticipate, several iterations of added complexity and improvements over a base algorithm result in a body of work which is not trivial to understand, both at the theoretical level and in terms of implementation. In this work, we present the following:

1. a tool, HCover, which boasts (at minimum) feature and performance parity with the ICover tool of Geffroy, Leroux and Sutre;
2. an explorable, well-documented and modular body of source code for HCover itself which may present a workable basis for further optimisations and improvements; and
3. the following text, a complete explanation of the end-to-end decision procedure in question[1], which is approachable for those familiar with Petri nets and VASS and may serve as an introduction to the pragmatics of state-of-the-art coverability techniques.

Use of Haskell

The tool HCover is implemented in Haskell. Haskell is a purely-functional, lazy, strictly typed programming language. As we shall see, each of these properties makes it an excellent candidate for implementing procedures such

[1]We acknowledge that this work does not present any novel theoretical improvements over the algorithm given in [63]—the contribution here is auxiliary to the algorithm itself and not technical in nature. It is nevertheless of value to the field for the reasons outlined above.

as this. Further, the Haskell language and ecosystem make significant use of abstractions: generalisations both of data and control flow. These abstractions may be seen as additional layers of complexity to a functional programming newcomer, but appropriate usage of these abstractions can vastly simplify the presentation of the algorithm's logic. If pains are taken to produce readable code, the result can be indistinguishable from pseudocode. Code segments given in this chapter will be provided as Haskell code; they will be annotated where appropriate to guide a reader unfamiliar with the language. For example, here is part of the source code of `HCover` which runs a continuous coverability query via an SMT solver:

```haskell
-- This is the type signature for the function.
checkCoverability
  :: PetriNet           -- The input net.
  -> Initial            -- The initial marking.
  -> Target             -- The target marking.
  -> Query SafetyResult -- The result of the query.
checkCoverability vas initial target = do
    let problem = constructProblem net initial target

    -- Run the encoded problem through the default SMT solver (z3)
    result <- runSMT problem
    case result of
        Nothing             -> return Safe
        Just counterexample -> return Unsafe
```

Figure 4.2: Constructing then running a coverability query via an SMT solver.

We see that, in the above code, the `checkCoverability` function first constructs an instance of the problem, given a Petri net, an initial marking, and a target marking. It then runs the problem instance through the SMT solver and returns `Safe` if the instance is not coverable, or `Unsafe` otherwise. It does not take a deep understanding of functional languages to follow the code as written, even with minimal comments from the author, since it closely resembles the form of pseudocode often seen in academic writing.

The first line of the code segment is the *type signature* of the function `checkCoverability`. This formally describes the relationship between the arguments and the result. Since Haskell is a strictly-typed programming language, if the function is provided improper arguments, the program will fail to compile. By encoding preconditions into the types, as we do here, we are providing compile-time guarantees that the code is structured properly. This affords some level of trust in the program's architecture which is not so easy to come by in other programming languages such as Python.[2] Haskell also guides

[2]Of course, Haskell is not magical; it is still possible to shoot oneself in the foot. But many of the typical failure modes seen in other languages—forgetting to initialise a variable, null references, improper casting—are simply not a concern in Haskell.

us towards writing shorter, more self-contained functions whose behaviour can be described by such type signatures.

`HCover` is able to take advantage of a number of automatic optimisations introduced by the Haskell runtime:

- There are a number of representations of sequences of values in Haskell. The two most frequently used are vectors (defined in the `vector` package) and lists (defined in the standard library, `base`). These represent something akin to arrays in typical languages and linked lists, respectively. Both of these types are amenable to a technique called *fusion.* If two consecutive operations would read each value of a vector or list in turn (for example, modifying each value in the structure and then checking if each value is equal to zero), then those two operations are *fused* into a single operation which will visit each value only once. In some settings, where a sequence of values would need be traversed repeatedly, this optimisation can drastically improve performance. Vectors, especially basis vectors, will be compared under $\leq$ many times in a run of this procedure, hence the relevance of this optimisation.
- Data structures such as `Map`s and `Set`s have both strict and lazy variants. When making use of their lazy variants, as is done in `HCover`, the runtime will only evaluate those elements that will be used in order to decide the output of a function. For example, if the procedure is comparing two vectors in $\mathbb{N}^{1000}$ by $\leq$, but it can determine inequality only by checking the first two components, then the other 998 will never be checked, and indeed might never be initialised at all. This has the capacity to vastly improve the performance of basis minimisation, which is one of the most computationally intensive parts of the procedure.

4.4.1 Implementation

We shall now describe the implementation of `HCover` in terms of control flow through the program.

The entrypoint of the executable is called `main`. Despite being purely functional, Haskell has a notation called “do” notation, which allows for the writing of procedural code by relying on the sequential properties of a sainted data type called `IO`. Working within the context of `IO` also gives us access to functions for interacting with the outside world; it forms the interface between functional purity of non-`IO` computations and the real world. The type of `main` is written as `IO ()`. This indicates that `main` returns no meaningful value—`()` being the unit type—and is built up of pure functions and computations defined over `IO`, to be evaluated sequentially. It begins:

```haskell
main :: IO ()
main = do
    (example:_) <- getArgs
    input       <- readFile example
    let res = parse readSpec example input
    ...
```

Figure 4.3: The entrypoint of `HCover`.

We take the first argument to `HCover` and use it as a filename to read the file with that name into a string called `input`. We then call a pure function, `parse`, which will interpret the string as an instance of the coverability problem. We shall skip any further uninteresting details related to input/output, error handling, and so on in order to focus on the algorithm itself on the assumption that the given problem instance is well formed.

The `HCover` procedure, like `ICover` and `QCover` is written with Petri nets as its data representation. This is in contrast to Karp and Miller implementations which typically use Vector Addition Systems. The explanations and code segments here will therefore be given in terms of Petri nets.

We start with a little preprocessing of the data. In particular we compute the pre-set, post-set and flow function for the provided net. The pre-set of a transition t is the set of places from which tokens are removed when t is fired; the post-set is the set of places that receive tokens when t is fired. The flow function prescribes, for each transition, the movement of tokens between places when that transition is fired. Refer to Section 3.5 for the formal definitions. These values will be used repeatedly during the coverability procedure so it is helpful to precompute them.

Our representation now consists of the net itself plus the precomputed data. The `PetriNet` data type is presented in Figure 4.4.

The types `P` and `T`, shorthand for “place identifier” and “transition identifier”

```haskell
data PetriNet = PN {
    placenames :: Map P String,
    flow       :: Map T [(Vector,Vector)],

    ps         :: [P],
    ts         :: [T],

    preset     :: Map T [P],
    postset    :: Map T [P],
}
```

Figure 4.4: The `PetriNet` data type.

respectively, are both represented by integers at runtime. This allows us to refer to places and transitions without the overhead of passing strings around the program. . Markings are stored densely as a `Vector` of integers—we store the values of all places, including zeroes. Trade-offs of this approach are discussed in Section 4.5 ("Alternative Representations of Vectors").

From here, we compute some inductive invariants of the net, and then proceed to the coverability loop.

Computing the Sign Invariant

The sign invariant, derived from data-flow sign analysis [29], is one of the inductive invariants used in the `ICover` algorithm. Intuitively, the sign invariant is an underapproximation of the set of places which will always be zero on a run starting from the given initial marking. Once we have this set, we can prune those markings where any of the places in that set have a nonzero number of tokens. The value of the sign invariant as an invariant depends on the structure of the net; a very interconnected net with an initial marking with many tokens is less likely to have a useful sign invariant than a sparse net with few places used.

The computation in `HCover` is exactly that described by Geffroy, Leroux and Sutre [63] adapted to the data types and conventions of our representation. To restate from [63, Section 6], and recalling the definition of an inductive invariant from Section 30, we represent the sign invariant by the maximal set of places Z such that

$$I_Z = \{M \in \mathbb{N}^d \mid \bigwedge_{p \in Z} M(p) = 0\}$$

is an inductive invariant.

Z is reached by a fixpoint computation. A fixpoint is a property of a given function, which here we shall call the *propagator*. The propagator is repeatedly called on its own result until the result is the same as the argument (a fixed point is found). Fixpoint computations are idiomatic in Haskell—indeed, most recursive patterns are implemented in terms of fixpoint computations under the hood. The function `converge` is a type of fixpoint computation which starts from an initial value. The definition is as given in Figure 4.5.

If the propagator does not converge to a value, then `converge` will never terminate. In this case, since the set of places is finite and the computed subset of places will grow at each step, we can guarantee convergence. Note that `converge` is polymorphic; it can operate over values of any type that we can compare for equality.

```haskell
converge :: Eq a => (a -> a) -> a -> a
converge f x =
  let x' = f x in
    if x == x'
      then x
      else converge f x'
```

Figure 4.5: The `converge` fixpoint function.

In this case we shall be using `converge` to successively grow a set of places. At each step, given the previous subset of places Q_k, we compute for each transition t the value $\text{prop}_t(Q_k)$ as defined below (for Petri net $\mathcal{N} = \langle P, T, f \rangle$).

$$\text{prop}_t(Q) = \begin{cases} \{q \in P \mid f(t,q) > 0\} & \text{if } \bigwedge_{p \in P \setminus Q} f(p,t) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

The intuition for the propagation is this. Call Q a set of “tags”. We start by tagging all the nonzero places in the initial marking (called Q_0). Then, for every transition t where all of the places $\mathbf{pre}(t)$ are tagged, we tag all the places in $\mathbf{post}(t)$ as well. The new larger set of tags is Q_1. We repeat this process until Q_{k+1} is no larger than Q_k for some k. The code corresponding to the computation of the sign invariant is as given in Figure 4.6.

```haskell
signInvariant :: PetriNet -> Marking -> [P]
signInvariant net m_0 = converge prop q_0
  where
    prop q = q `union` (unions $ map (prop_t net q) (ts net))
    q_0    = filter (>0) m_0
```

Figure 4.6: The computation of `signInvariant`.

The result of this computation is Q_k, an overapproximation of the set of places which could ever become nonzero as the result of a sequence of transitions starting from the initial marking. Hence, the complement of Q_k with respect to P is an underapproximation of the set of places which will *always* be zero. This is the set Z.

Recall from Section 30 that an invariant is a downward-closed set of markings. To turn Z into into a downward-closed set which we can use to prune the search space, we construct $\downarrow(Z)$ as follows:

$$\downarrow(Z) = \{M \mid p \in Z \rightarrow M(p) = 0\}.$$

Any marking that is not part of $\downarrow(Z)$ is guaranteed to be uncoverable from the initial marking, and so we can at every step of the backwards search prune any markings which are not in $\downarrow(Z)$.

Shortcutting via Continuous Coverability

As discussed previously, we can view continuous coverability as a relaxation of coverability. In the continuous setting, coverability checking can be done in polynomial time. At the beginning of a run of `HCover`, we check whether the instance is $\mathbb{Q}$-coverable. If not, we can immediately say that the instance is not coverable in the integer setting either.

In `HCover` we run this query through an SMT solver. The SMT solution used is an excellent library called `SBV` (**S**MT **B**ased **V**erification). `SBV` introduces an embedded domain-specific language for constructing and evaluating queries to an SMT solver. It also abstracts over a number of different solvers, allowing the developer to remain agnostic; by default it will run the first solver it finds on the user's machine, but the developer can override this to choose a particular solver which is known to be better for some use case. (This technique will later be used in Chapter 5 to great effect.)

The high-level code for running the coverability query was given as an example in Figure 4.2. The procedure itself is as given by Blondin et al. [13, Section 4]. We shall not present every step of the computation of rational reachability here, but as a representative sample, the global $\mathbb{Q}$-reachability relation $\Phi_{\mathcal{S}}$ is defined as follows. In the terminology of [13], w and x are the initial and final markings; y is the mapping from transitions to $\mathbb{Q}$ representing the amount each is fired; $\mathcal{N}$ is the net itself; Φ_{eqn} is the state equation over rationals; $\Phi_{fs}^{\mathcal{N}}$ captures valid firing orders for pre-sets in $\mathcal{N}$; and $\Phi_{fs}^{\mathcal{N}^{-1}}$ captures the same for post-sets. Then:

$$\Phi_{\mathcal{S}} \overset{\text{def}}{=} \exists y \ : \ \Phi_{eqn}(w,x,y) \wedge \Phi_{fs}^{\mathcal{N}}(w,y) \wedge \Phi_{fs}^{\mathcal{N}^{-1}}(x,y).$$

The code corresponding to $\Phi_{\mathcal{S}}$ is given in Figure 4.7. The types `SReal` and `SBool` are purely symbolic variables which can be used to construct first-order logic sentences. Note that, due to the functionally pure nature of Haskell, evaluating $\Phi_{\mathcal{S}}$ (or any other function in the `Symbolic` context provided by `SBV`) is *not* the same as evaluating or computing the result of the query itself. Instead, by calling these functions, we are constructing a pure representation of the entire problem in the first order logic. In order to evaluate the problem instance, we use the function

```
runSMT :: Symbolic a -> IO a
```

which translates from the `SBV`'s intermediate representation to the solver's native language (`SMTLib`), then interacts with the outside world (in this case, a solver on the user's machine) to run the query through the solver and retrieve the result.

```
-- Encoding rational reachability into SMT.
phi_s :: PetriNet -> [SReal] -> [SReal] -> Symbolic SBool
phi_s n w x = do
    -- create the vector representing the image of some run
    y <- mapM exists ["t_" ++ show t | t <- ts n]

    -- Step 1: The rational state equation
    pe  <- phi_eqn n w x y
    -- Step 2: Ensuring firing set membership
    pf  <- phi_fs n "fwd" w y
    -- Step 3: Ensuring REVERSE firing set membership
    pf2 <- phi_fs (inverse n) "rev" x y

    return (pe &&& pf &&& pf2)
```

Figure 4.7: Symbolic representation of $\Phi_{\mathcal{S}}$ with `SBV`.

The Coverability Loop

With all the static computations complete, it remains to run the backwards coverability algorithm. The implementation follows the same process as described in Algorithm 2 (see page 36) with a small variation: before taking the upward-closure of the new set of predecessors, we remove those predecessors in our existing set U and those that fall outside of the downward-closed invariant. (In our case, we use the singular downward-closed invariant derived from sign analysis.) If all the new predecessors are eliminated by this process, then we know that our set has not grown, and we terminate with a result of `Uncoverable`. If we do get new predecessors then they are added to the set in the usual way and we continue to loop.

The looping code is given in Figure 4.8, with user-facing output removed for simplicity. There is a clear correspondence between lines 6-10 of the `ICover` algorithm [63, Section 4] and the functionality of `covLoop`.

4.4.2 Results

The source for `HCover` is online and available on the source code host GitHub.[3] It comprises around 550 lines of Haskell code—fewer than `QCover` and less than one half of the `ICover` tool which implements the same procedure. The code is divided across around a dozen modules, each of which describes a single piece of the algorithm, such as `SignAnalysis` for computing inductive invariants based on sign analysis and `UCSet` for computing and minimising upward-closed sets.

While there is always room for improvement, it is believed that the code base of `HCover` meets the intended goals: to serve as a end-to-end explanation

[3]`https://github.com/dixonary/hcover`

```
covLoop :: [Marking] -> Bool
covLoop b =
  -- "contains" means contained within the upward closure of b
  if b `contains` init
    then True -- We found init in our set!
    else do
      -- Compute new predecessors
      let n = pred net b

      -- remove any element "m" of n which is in b
      -- or is not in I_z
      let p = filter prune n
        where prune m = not (b `contains` m) && m <= i_z

      -- if newPred is empty, return False
      -- otherwise union p into b
      if null newPred
        then False
        else covLoop (basis (b `union` p))
```

Figure 4.8: The backwards coverability loop in `HCover`.

of the procedure itself which is relatively approachable to newcomers, and to serve as a potential base for future improvements over the algorithm. Potential such improvements are described at the end of this chapter.

Benchmarks

`HCover` was initially benchmarked against two other tools: `QCover`[4] and `MIST`. `MIST` is a competitive solver whose primary feature is the use of Interval Sharing Trees to represent (upward-closed sets of) markings, as introduced in [58] and formalised into the tool presented in [60]. `MIST` is also the originator of the `.spec` format that has since been taken by several coverability checkers as a de-facto standard, hence it serves as a natural choice of tool to use as a benchmark.

The results of initial benchmarking are presented in Table 4.1.

Benchmark Set	`MIST`	`QCover`	`HCover`
MIST (16)	**13**	10	12
Soter (50)	0	**50**	**50**

Table 4.1: The number of coverability instances decided by `MIST`, `QCover` and `HCover`. The number in parentheses is the total number of instances used from that class.

The set of benchmarks used is a subset of those provided alongside the

[4]Unfortunately there were some technical difficulties in getting the `ICover` tool to function, due to changes over time in the Python ecosystem. However, as noted previously, the performance characteristics of `ICover` and `QCover` are broadly comparable.

MIST tool.[5] The results were computed on an early-2017 MacBook Pro with a 2.3GHz dual-core Intel Core i5 CPU and 8GB of LPDDR3 RAM. The timeout was configured to 1000 seconds. HCover is able to decide every instance that QCover decided and a couple of additional instances from the MIST set. In the majority of cases, HCover outperforms QCover significantly. In those cases where QCover outperforms HCover, the instances are usually very small; the difference is attributed to the additional overheads induced in parsing the specification in HCover, which was not a target of optimisations.

Predictably, the MIST tool is very capable of solving the MIST benchmarks, though it struggles with the magnitude of the SOTER instances. While it solves only slightly more than HCover within the 1000-second per-instance time limit, it tends to solve these instances much faster. It is conjectured that the interval sharing tree format is a major factor in this speedup as performance breakdowns suggest that basis computation forms the bulk of time taken in these cases in HCover.

4.5 Potential Further Work

We can safely say that HCover is at least at parity with its progenitors in terms of performance, and meets its goals of being a readable and extensible code base. However, HCover is a relatively naïve and unrefined implementation of the ICover algorithm; there are plenty of opportunities for improvements, both in the existing code base and through theoretical breakthroughs that have occurred since the original ICover implementation was posited. It remains to see whether any or all of these improvements offer a significant boost to run time performance. Some lines of enquiry are laid out here. Some of these methods are applicable to HCover, while others are improvements that may be incorporated into other solvers such as the accelerated implementation of Karp and Miller that appears in Section 44.

Pruning and Refinement In their conclusion to [13], Blondin et al. conjecture that a tool implementing every acceleration known to date would be capable of solving every benchmark in the literature within 2000 seconds (around half an hour). One such improvement for forwards coverability may be a variant of Finkel's pruning criterion.

In 1991, Finkel [50] presented an aggressive and highly effective method of pruning the Karp-Miller tree. Informally, while one constructs the Karp-

[5]The subset chosen corresponds to those that can be parsed according to the definition of the MIST net specification—several instances do not fit the grammar given and therefore would require special handling which has not been undertaken here.

Miller tree, one prunes the tree by deleting all previously seen subtrees which are rooted at any vector smaller than the current node. The intention of this pruning criterion was to ensure minimality of the generated tree, since (so claimed Finkel) every subtree of vector v would naturally be smaller in some sense than every subtree of v', for $v' > v$.

Unfortunately, it was shown by Geeraerts, Raskin and Van Begin in 2007 [61] that there is an intricate and not-easily-corrected flaw in this approach, which causes it to underapproximate the minimal coverability set due to overly aggressive pruning. They then go on to provide a more complex but ultimately effective strategy for computing the minimal coverability tree.

We note the following:

- a coverability procedure does not require the computation of a *minimal* coverability set or tree;
- a minimal coverability tree may be recovered from a non-minimal one, if necessary; and
- pruning previously-computed parts of the tree is a way of minimising the tree but will not necessarily reduce the runtime of the algorithm, since the parts being pruned have already been computed.

Given these remarks, we posit a considerably simpler approach to computing the coverability tree, which does not rely on the overly aggressive pruning of Finkel's acceleration. Firstly, we construct the tree in depth-first order, rather than breadth-first.[6] Secondly, while constructing the tree in the standard approach of Karp and Miller, we maintain a set M of the maximum vectors seen. (Using the terminology of the previous section, we consider this a basis of the downward-closed set of all known-coverable vectors.) If, during our depth-first construction of the complete Karp-Miller tree, we visit a node v such that $v \in M$, then we may immediately discard it, since (due to the depth-first order) all vectors coverable from v will *already* have been covered in the partially evaluated tree.

Preliminary evaluation of this pruning criterion seems to be fruitful. Encoding the simplified Karp-Miller implementation into a tool and testing it separately remains to be done, but it is suspected that this approach is competitive with other Karp-Miller based algorithms, especially in cases where the entire tree need be explored. If this does prove to result in a significant speed-up then an implementation and formal explanation may be of publishable significance.

[6]The order of computation is not relevant for Finkel, Geerarts, Raskin and Van Begin, but it is required for correctness in the alternative pruning technique outlined here.

Alternative Representations of Vectors There are many ways to represent a list of integers. Two classes of representations can easily be identified:

- **Sparse vectors** store only the *support* of the vector—those components whose values are nonzero—and their associated value. The representation is typically a mapping from component index to value, with absent components having their value taken to be zero. Taking a `Map` as the typical implementation of such a mapping, sparse vectors have $O(\log n)$ lookup and modification in the size of the support of the vector[7]. Furthermore, there are few fusable operations like "zipping" (running a function pairwise across two structures), which means essential componentwise vector operations like comparison under $\leq$ may be unoptimised and slow.

- **Dense vectors** (or **total vectors**) are a complete association of component to value. The typical representation is an array-like `Vector`. These have well-optimised componentwise operations which makes for fast comparisons under $\leq$ and they benefit from vector fusion in ways that maps cannot. However, the obvious downside of total vectors is that they are of linear size in the dimension of the instance. Memory may therefore become the limiting factor for any instances where the basis grows very large, or where the dimensionality is high.

At present, `HCover` uses total vectors to represent `VAS` vectors. Given the timeout of 1000 seconds, this has not posed issues. However, larger instances—those more likely to take more than 1000 seconds to solve—may have larger dimension or larger bases and therefore `HCover` may potentially suffer an outsized slowdown. One alternative may be to statically inspect the shape of the input instance: if we determine that the dimension is very high, or the number of zeroes is likely to be high, then use a sparse vector representation; otherwise we a total vector representation. This approach may offer the best chance of efficiently solving both smaller and larger instances.

However, there is an alternative formulation, which has not yet been seen in any tool, but which may represent an effective strategy, combining the benefits of total and sparse vectors.

Consider the following strategy. Take the infinite sequence of prime numbers $(2, 3, 5, \ldots)$, and the vector $v = (v_0, v_1, v_2, \ldots)$. Compute the integer $I(v) = 2^{v_0} \cdot 3^{v_1} \cdot 5^{v_2} \cdot \cdots$. We call $I(v)$ the *prime encoding* of the vector v. In code:

```
primes :: [Integer]
```

[7]The underlying representation of the canonical `Map` data type is a binary tree.

```haskell
primes = [2, 3, 5, 7, 11, ...]

-- zipWith applies a binary function componentwise.
primeRep :: [Integer] -> Integer
primeRep vec = product (zipWith (^) primes vec)
```

Using this encoding, we get that `primeRep [0,0,0]` is equal to 1, `primeRep [0,0,1]` is 5, and `primeRep [0,1,2]` is 75. This is, in essence, the inverse of computing the prime factorisation of a number.

Proposition 4.1. *We have $v \leq v'$ if and only if $I(v') \textbf{ mod } I(v) = 0$.*

Proposition 4.1 relates the prime encoding of two vectors to the ordering of those two vectors under $\leq$. If $I(v') \textbf{ mod } I(v) = 0$ then $I(v')$ is some multiple of $I(v)$; so there is some $k \in \mathbb{N}$ such that $I(v') = k \cdot I(v)$. The fundamental theorem of arithmetic (due to Gauss) guarantees that k has a unique prime factorisation; moreover since k is itself a factor of $I(v')$, its factorisation will be composed only of primes not exceeding the largest prime factor of $I(v')$. If we write the prime factorisation of k as a vector w in the same dimension as v and v', then the value of each component of that representation is the exponent of the corresponding prime. If $I(v') = k \cdot I(v)$, then $v' = w + v$. Since all the components of w are clearly nonnegative, it must hold that $v \leq v'$. The same logic holds in reverse. An example of the arithmetic involved is given in Table 4.2.

	$I(v')$	$I(v)$	k
Prime encoding	2250	50	45
Prime factorisation	$2^1 \cdot 3^2 \cdot 5^3$	$2^1 \cdot 3^0 \cdot 5^2$	$2^0 \cdot 3^2 \cdot 5^1$
Vector representation	$[1, 2, 3]$	$[1, 0, 2]$	$[0, 2, 1]$

Table 4.2: Sample values showing the relationship between a vector and the prime encoding of that vector.

Proposition 4.1 means that we need only perform a single arithmetic operation in order to check $\leq$, regardless of the dimensionality of the vectors in question. Of course, the time taken for that operation will still depend on the size (dimension and componentwise magnitude) of the vectors. One would be forgiven for thinking that this approach will obviously be slower than determining $\leq$ based on a precomputed factorisation. However, this seems not to be the case in reality.

By default, Haskell represents large integer values using a library called GMP (the GNU Multiple Precision Arithmetic Library)[8]. GMP makes use of machine integers when working with small values and its own custom implementation of larger values. GMP has highly optimised assembly code for each

[8] `https://gmplib.org/`

CPU architecture in order to minimise the number of instructions required to perform operations such as modulo, even on extremely large values.

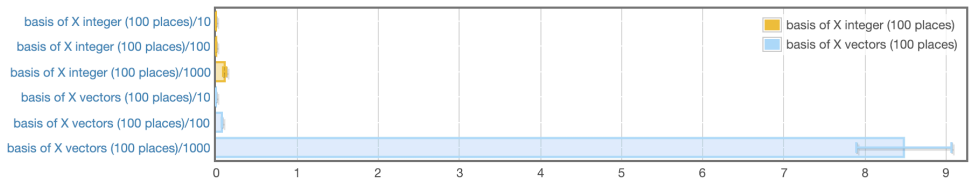


Figure 4.9: Speed of operations over values in dense vector representation (blue) versus integer encoding (yellow).

A rudimentary but fair test was constructed to verify that this approach is feasible. A number of vectors was generated, in dimension 100 (enough to encode moderately-sized coverability instances). The value of each component was uniformly chosen from $[0, 1000]$ (higher than the values typically seen in Vector Addition Systems, but a useful test of how the integer encoding can cope with exponents). The minimal basis was computed for each set, requiring at least $O(n^2)$ operations. Figure 4.9 shows the results over sets of 10, 100 and 1000 vectors.

The outcome of this synthetic test suggests a speedup of approximately $100\times$ on vectors on up to 100 places. Part of this speedup is likely due to the overhead of allocating dense vectors, but profiling indicates that the bulk of time spent is in performing the comparisons. Given the amount of allocation that is required by the standard procedure anyway, it would be unsurprising to see commensurate speedup in practice.

This technique is not immediately applicable to all coverability approaches. For example, the standard Karp-Miller approach requires keeping track of "ω" values—places whose value is unbounded due to loops in the VAS. Finite multiplications of primes would not allow for such a construction. However, if a separate set of "ω-components" were maintained, this approach might have promise.

There is likely some point at which the big-integer approach becomes untenable. For example, the computation

```
999999^1000001 `mod` 1000000^1000000 == 0
```

takes over a second to calculate on the test machine; a sparse vector representation would be able to compute $\leq$ almost instantly here. If it were possible to characterise the inflection point between effectiveness of methods, we could switch between representations on the fly. (Note that this example would require a million places and a million tokens in the millionth place; such a shape of net is unlikely to appear in the wild). It might also be possible to statically arrange nets so that places most likely to grow large appear earlier in the prime representation, to reduce the size of induced integers.

Note also that this approach is only useful for "direct" solutions which explore the state space, and not to CEGAR (Counterexample-Guided Abstraction Refinement) based approaches. While this improvement could be considered an implementation detail, the resulting speedup may be significant enough to warrant considerable investigation for practical solving.

GPU based solving To date, there have been no known attempts to solve coverability in a massively parallel way—that is, by reducing subproblems to a form distributable across hundreds or thousands of cores at once. Since the development of `HCover`, a Haskell library called `accelerate` [27] has reached maturity. This library includes an embedded domain-specific language for representing operations over arrays (equivalently, vectors) in such a way that they can be offloaded to the many computational cores of a modern GPU on-the-fly. The library has already been used for highly performant implementations of programs in a variety of domains including hydrodynamic simulations[9,10], image manipulation[11], and sudoku solving[12]. Vector Addition Systems would appear to be a natural fit; it remains to be seen if the computationally intensive steps of the algorithm, such as basis minimisation, can be parallelised sufficiently.

[9]`https://github.com/tmcdonell/lulesh-accelerate`
[10]`https://github.com/GeneralFusion/gpu-fv-mhd`
[11]`https://hackage.haskell.org/package/patch-image`
[12]`https://github.com/dpvanbalen/Sudokus`

Chapter 5

Petri Net Reachability & KReach

With a discussion of the coverability problem on Petri nets and Vector Addition Systems complete, we shall now move on to reachability. Coverability and reachability are two intrinsically related problems. As with coverability, we may formulate the reachability problem both over Petri nets and VASS:

Definition 5.1. The *reachability problem for Petri Nets* asks, given a Petri net $\mathcal{N} = \langle P, T, \mathcal{F} \rangle$, an initial marking M_0, and a target marking M, whether there is any run $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M$ in $\mathcal{N}$.

Definition 5.2. The *reachability problem for Vector Addition Systems with States* asks, given a VASS $V = \langle Q, T \rangle$, an initial configuration c_0, and a target configuration c, whether there is any run $c_0 \xrightarrow{t_0} c_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} c$ in V.

The difference between reachability and coverability is this: while coverability queries permitted the net to visit any marking $M' \geq M$, reachability requires us to visit exactly the marking M.

Despite the minimal change, the difference in difficulty between the two problems is immense. While coverability is known to be EXPSPACE-complete, reachability has only recently been determined to be complete for ACKERMANN [30, 102].

5.1 Why Reachability?

Just as the coverability problem captures many safety properties, reachability is able to capture liveness properties of systems [75]. Liveness is a characteristic of transitions, such that a transition t is *live* if and only if, from every marking M reachable from the initial marking, there is a run to any marking M' where

t is enabled in M'. *Structural* liveness extends the concept of liveness to the entire net. Liveness can be used to deduce the risk of deadlocking—reaching some configuration where it is no longer possible to continue firing (some set of) transitions. One example of the benefit of liveness (and, as a corollary, deadlock-freeness) is to locate bugs in concurrent systems that share resources.

It is reasonable to view reachability as a strictly more powerful version of coverability. Analogous to $\mathbb{Q}$-coverability discussed in Section 28, we can reduce a VASS coverability query to a reachability query only by adding states and transitions and modifying the target configuration.

Proposition 5.1. *The following are equivalent:*[1]

1. *The coverability problem for a* VASS $\mathcal{S} = \langle Q, T\rangle$, *with initial configuration* c_0 *and target* (q, v)*;*
2. *The reachability problem for* $\mathcal{S}' = \langle Q \cup \{\triangledown\}, T'\rangle$, *with initial configuration* c_0 *and target* $(\triangledown, \vec{0})$, *where*

$$T' = T \cup \{(q, -v, \triangledown)\} \cup \{(\triangledown, \{i \mapsto -1\}, \triangledown) \mid i \in \{1, \dots, d\}\}.$$

Proof. First we note that the transition $(q, -v, \triangledown)$ can be fired from some configuration (q, v_k) if and only if $v_k \geq v$ (i.e. if and only if (q, v) is covered). Secondly we note that $(\triangledown, \vec{0})$ can be reached if and only if $(q, -v, \triangledown)$ can be fired. The latter is thanks to the newly added set of transitions which do nothing but reduce each component by 1, which renders $(\triangledown, \vec{0})$ reachable from any configuration with state $\triangledown$. The conjunction of these two facts completes the proof. $\square$

This specific translation shall be used later in this chapter to produce reachability problem instances from coverability instances.

5.2 Kosaraju's Reachability Algorithm

We shall focus our attention on one method of deciding the reachability problem, due to Kosaraju [90], building on the work of Mayr [107] and Sacerdote and Tenney [140]. The algorithm has since been worked on further by Lambert [95]. This lineage has resulted in the combined work being known as the *KLMST algorithm*, and the main component of the algorithm as the *KLMST decomposition*. However, in this work we shall focus on the Kosaraju's constructions rather than any simplifications and variations since. We shall refer to *Kosaraju's algorithm* when talking about the treatment from 1982.

[1]This is essentially folklore—a well-known and established truth, but not easily traced back to an initial source.

We believe that prior to this work there were no known implementations of Kosaraju's algorithm—or indeed any exact reachability procedure for Petri nets. Some very recent work of Amat, Zilio and Hujsa [6] offers a capable semi-decider for reachability based on statically derived properties of the net. In a similar vein, Blondin, Haase and Offtermatt [15] have produced a semi-deciding tool which is effective in practice and uses a lightweight overapproximation of reachability to guide traditional search methods like A^*. However, being optimised towards reachability and not unreachability, both of these methods are liable to reach dead ends or run indefinitely on particularly thorny or degenerate shapes of nets.

We believe the relative lack of implementations of full deciders for reachability to be due to two primary factors. Firstly, the algorithm was not believed to be particularly effective (in terms of performance)—as with many decidability proofs, the runtime speed of the construction was not a concern. The second reason is that Kosaraju's algorithm is notoriously complex; it is the subject of an entire book by Reutenauer [136], since translated into English [135], and more recently presented in a novel and readable format with contemporary notation by Lasota [96].

Continuing in the same spirit as Chapter 4, we intend this work to be meaningful both in terms of practical value—as a tool which can be used—but also as a piece of pedagogic value for those new to decision procedures for Petri nets. To that end, we offer the following contributions in this chapter.

- We present and describe a tool, `KReach`, which we believe to be the first complete implementation of a decision procedure for the general Petri net reachability problem.
- Noting that the reachability algorithm is generally regarded with trepidation due to its complexity—both in terms of worst-case runtime and the impenetrability of the decision procedure for newcomers—we offer an accessible implementation of Kosaraju's algortihm, which can be used as an in-depth learning aid.
- The code base of `KReach` is designed in a modular and extensible way, which is conducive to development of future modifications. To evidence this we implement some examples of minor theoretical improvements over Kosaraju's original algorithm.
- We provide a full suite of libraries which aid programming with Vector Addition Systems (with States) in the Haskell programming language.

5.2.1 An Overview of the Algorithm

We shall begin with an exposition of the structure and logic of Kosaraju's algorithm. Unlike the coverability procedure described in Chapter 4, here we shall stick to Vector Addition Systems with States throughout—Kosaraju's algorithm does not make use of Petri nets.

Let us assume for the remainder of this section that we wish to determine reachability for a VASS $V = \langle Q, T\rangle$, with initial configuration $c_0 = (q_0, v_0)$ and final configuration $c = (q, v)$.

Kosaraju's algorithm is a CEGAR (CounterExample-Guided Abstraction Refinement) based procedure. The counterexamples are derived from a holistic predicate called θ. The abstraction to be refined is an augmented variant of VASS, called GVASS (short for Generalised VASS): a sequence of VASSs which have been augmented with a large amount of metadata. We will interest ourselves in runs over these GVASSs, which can be considered as a run over all of the individual components in turn.

Definition 5.3. A GVASS is a pair $\boldsymbol{G} = \langle \boldsymbol{C}, \boldsymbol{A}\rangle$, where

- $\boldsymbol{C}$ is a nonempty sequence $(\mathcal{C}_1, \ldots, \mathcal{C}_\ell)$ of *components*, where each component is a triskaidecuple[2] of the form

$$\mathcal{C}_i = (d, Q_i, T_i, q_i, q_i', R_i, r_i, C_i, U_i, C_i', U_i', v_i, v_i');$$

- $\boldsymbol{A}$ is a sequence $(a_1, \ldots, a_{\ell-1})$ of *adjoinments*, where each adjoinment is of the form $a_i = (q_i', z_i, q_{i+1})$ for $z_i \in \mathbb{Z}^d$.

It behooves us to present a brief run-down of the contents of a component. Recall that each component is in effect its own VASS, with additional metadata attached. The values d, Q_i and T_i are recognisable from VASS:

- d is the dimension of the VASS.
- Q_i is the set of states.
- T_i is the set of transitions.

The additional metadata attached to a GVASS component represent successive attempts to constrain the portions of runs that take place through that component. In particular, the run can be unconstrained, constrained, or rigid (fixed) in any number of coordinates of the vectors at the start and/or end of the component. The initial and final states are always fixed.

[2]A 13-tuple.

- q_i/q'_i are the initial/final states.
- R_i is the set of rigid coordinates.
- $r_i : R_i \to \mathbb{N}$ is the mapping of rigid values.
- C_i/C'_i are the sets of constrained initial/final coordinates.
- U_i/U'_i are the sets of unconstrained initial/final coordinates.
- $v_i : C_i \to \mathbb{N}/v'_i : C'_i \to \mathbb{N}$ are the initial/final constrained vectors.

The intuition behind the three categories of coordinates is as follows.

- If a coordinate is marked as rigid in component $\mathcal{C}_i$ (it is in R_i) then the value of that coordinate is not permitted to change on transitions inside of $\mathcal{C}_i$. Instead, its fixed value is written in r_i, and any changes corresponding to those moves which would have occurred in the net are instead added onto the adjoinment that follows the component.

- If a coordinate is initially or finally constrained in $\mathcal{C}_i$ (it is in C_i or C'_i, respectively) then we know the value of that coordinate as a run on the GVASS enters or leaves the given component. That known value is tracked in v_i or v'_i. These constraints will be used to identify suitable runs according to the θ condition to be described shortly. A value which is initially *and* finally constrained will often be made rigid.

- If a coordinate is initially or finally unconstrained in $\mathcal{C}_i$ (it is in U_i or U'_i) then its value is allowed to change arbitrarily on runs through the component. Parts of the θ condition will be explicitly looking at the behaviour exhibited by unconstrained coordinates and violations of θ represent opportunities to constrain them.

Definition 5.4. A GVASS is *well-formed* if all of the following hold.

- *The dimension is constant.* d is the same for all $i \in \{1, \ldots, \ell\}$.

- *All coordinates are rigid, constrained, or unconstrained.*
  For all $i \in \{1, \ldots, \ell\}$:

  - $R_i \cup C_i \cup U_i = R'_i \cup C'_i \cup U'_i = \{1, \ldots, d\}$; and
  - $|R_i| + |C_i| + |U_i| = |R'_i| + |C'_i| + |U'_i| = d$.

- *Adjoinments are coherent.* For all $i \in \{1, \ldots, \ell - 1\}$ and $x \in \{1, \ldots, d\}$, if $x \in C'_i \cap C_{i+1}$ then $z_i[x] = v[i] - v'[i]$. (Recall that z_i is the vector labelling the adjoinment a_i between C_i and C_{i+1}.)

We shall only interest ourselves with well-formed GVASSs; and moreover every decomposition will leave us with a well-formed GVASS. Let us briefly describe how we may "lift" our reachability instance over VASS into a GVASS form.

Definition 5.5. For any given VASS $V = \langle Q, T \rangle$ in dimension d, initial configuration $c_0 = (q_0, v_0)$ and target configuration $c = (q, v)$, there is a *lifted* GVASS $\boldsymbol{G}(V, c_0, c) = \langle (\mathcal{C}_1), \boldsymbol{A} \rangle$, where

$$\mathcal{C}_1 = (d, Q, T, q_0, q, \emptyset, \emptyset, Q, \emptyset, Q, \emptyset, v_0, v); \text{ and}$$
$$\boldsymbol{A} \text{ is the empty sequence.}$$

The functionality of Kosaraju's algorithm is built around a predicate—the θ condition—over GVASS, and a notion of decomposition. Both will be covered in more detail shortly. The result tying them together is as follows.

Proposition 5.2. *Given a* VASS $V = \langle Q, T \rangle$ *in dimension* d*, configuration* c *is reachable from configuration* c_0 *if and only if some* GVASS *in the decomposition tree rooted at* $\boldsymbol{G}(V, c_0, c)$ *satisfies* θ*.*

Violations of θ act as counterexamples that induce particular decompositions. Eventually, either θ will hold for some such decomoposition (the target is reachable), or there will be no more such decompositions that can be performed (the target is not reachable).

GVASS **Notation**

In order to give suitable examples of GVASS in the remainder of the chapter, we shall extend our diagrammatic VASS notation to represent components and the various constraints that are imposed on those components. Each component is itself drawn as a VASS and is contained in a box. States and transitions are labelled as usual. Note that the component has an inbound edge to q and an outbound edge from r which enter and leave the component with values denoted by $\langle$algebraic brackets$\rangle$. These represent the initial and final vectors. Unconstrained initial/final coordinates will be marked by an asterisk $*$. Rigid coordinates will be marked with an overline and will necessarily have the same value in the initial and final vector for a component. Adjoinments shall be denoted like transitions except they will leave one component at its final state and enter another at its initial state, and will be marked with a dashed arrow.

Example 5.1. Consider the GVASS diagrammed in Figure 5.1. In the left component, both coordinates are initially constrained; the first coordinate is finally unconstrained and the second is finally constrained to 500. In the right component, the first component is initially unconstrained and finally set to 0; the second component is rigid at 499.

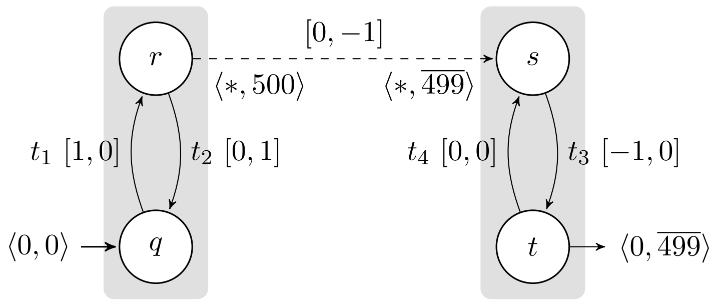


Figure 5.1: An example GVASS with two components, and rigid, constrained and unconstrained coordinates.

We can formally write the given GVASS as $\langle(\mathcal{C}_1, \mathcal{C}_2), ([0,-1])\rangle$ where $d = 2$ and the two components are defined as follows.

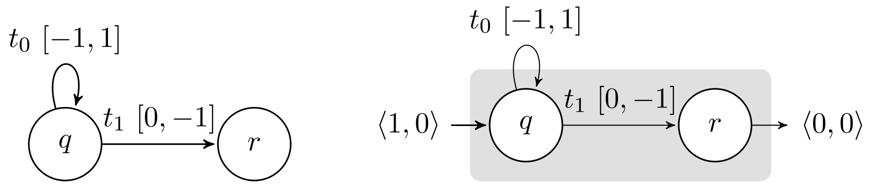
$$\begin{aligned}
Q_1 &= \{q, r\} & Q_2 &= \{s, t\} \\
T_1 &= \{t_1, t_2\} & T_2 &= \{t_3, t_4\} \\
q_1 &= q & q_2 &= s \\
q_1' &= r & q_2' &= t \\
R_1 &= \emptyset & R_2 &= \{2\} \\
r_1 &= \emptyset & r_2 &= \{2 \mapsto 499\} \\
C_1 &= \{1, 2\} & C_2 &= \emptyset \\
U_1 &= \emptyset & U_2 &= \{1\} \\
C_1' &= \{2\} & C_2' &= \{1\} \\
U_1' &= \{1\} & U_2' &= \emptyset \\
v_1 &= \{1 \mapsto 0, 2 \mapsto 0\} & v_2 &= \emptyset \\
v_1' &= \{2 \mapsto 500\} & v_2' &= \{1 \mapsto 0\}
\end{aligned}$$

As an aside, a run does exist through this GVASS which meets all the requirements specified in the metadata—identifying such a run is left to the reader as an exercise.

Example 5.2 (Lifting)**.** Consider a VASS

$$V_{\mathsf{ex}} = \langle\{q, r\}, \{(q, (-1, 1), q), (q, (0, -1), r)\}\rangle.$$

We wish to decide reachability from $(q, (1, 0))$ to $(r, (0, 0))$. The diagrams for V_{ex} and $\boldsymbol{G}(V_{\mathsf{ex}}, (1, 0), (0, 0))$ are given in Figures 5.2a and 5.2b respectively.



(a) Example VASS V_{ex}. (b) Lifted GVASS $\boldsymbol{G}(V_{\mathsf{ex}}, (1,0), (0,0))$.

Figure 5.2: An example VASS, V_{ex}, and its lifting into a GVASS.

We would formally write $\boldsymbol{G}(V_{\mathsf{ex}}, (1,0), (0,0))$ as $\langle(\mathcal{C}_{\mathsf{ex},1}), \boldsymbol{A}_{\mathsf{ex}}\rangle$, where

$$\mathcal{C}_{\mathsf{ex},1} = (2, \{q, r\}, T_{\mathsf{ex},1}, q, r, \emptyset, \emptyset, \{q, r\}, \emptyset, \{q, r\}, \emptyset, (1, 0), (0, 0))$$
$$T_{\mathsf{ex},1} = \{(q, (-1, 1), q), (q, (0, -1), r)\}$$

$\boldsymbol{A}_{\mathsf{ex}}$ is the empty sequence.

The θ Condition

Recall that the lifted GVASS corresponding to our reachability query (henceforth referred to as $\boldsymbol{G}$) includes the initial and final configurations as part of its construction. Intuitively, the θ condition over $\boldsymbol{G}$ represents the presence or absence of a particular kind of run through $\boldsymbol{G}$.

We may view the θ as the conjunction of two other predicates. The first (θ_1) is a global condition on $\boldsymbol{G}$; the latter (θ_2) is a condition which must hold within each component. θ_1 is a slight relaxation of reachability, such that the set of GVASSs for which θ_1 and θ_2 both hold are those in which reachability is guaranteed. The correctness of this procedure with respect to the θ condition is briefly discussed in Section 5.2.5.

The θ_1 Condition. The first condition is interested in the presence of so-called *unbounded pseudo-runs* through $\boldsymbol{G}$. A *pseudo-run* is a relaxation of a run, where vector coordinates are allowed to vary over $\mathbb{Z}$ rather than $\mathbb{N}$. We wish for these pseudo-runs to be *unbounded*: they must be able to use every transition in every component an unbounded number of times, and in doing so attain unboundedly large values in every unconstrained coordinate. Any constraints in values in the initial and final positions of any component must also be met by such a pseudo-run.

For example, consider the simple GVASS given in Figure 5.3.

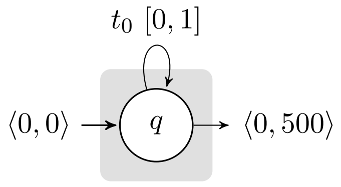


Figure 5.3: A one-component GVASS which does not satisfy θ_1.

While there is a clear pseudo-run (firing t_1 500 times forms a valid run), it is not unbounded—we cannot fire t_0 an unbounded number of times while also completing the run. So the GVASS in Figure 5.3 does not satisfy θ_1. By contrast, the GVASS in Figure 5.4 *would* satisfy θ_1, as we may (for example) fire t_0 n times and fire t_1 $(n-500)$ times, for any $n > 500$. Note that any interleaving of those firings would form a valid pseudo-run, including interleavings in which the second coordinate goes below zero, since we do not care about maintaining nonnegativity.

Observe that the first coordinate stays at zero for the entire pseudo-run. This is not a violation of θ_1, since we only need to reach unbounded values in unconstrained coordinates, and in the GVASS in Figures 5.3 and 5.4 all coordinates are constrained. The values attained in either coordinate are irrelevant except for satisfying the initial and final constraints; the important thing (for this toy example) is that the transitions are fireable an unbounded number of times.

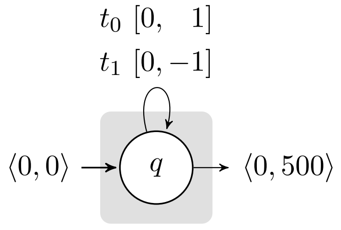


Figure 5.4: A one-component GVASS which does satisfy θ_1.

The θ_2 Condition. While θ_1 is more interested in the unbounded firability of transitions, θ_2 is solely interested in being able to pump (unconstrained) coordinates arbitrarily high via some cycle within each component.

The θ_2 condition is a property that must be maintained within each component of $\boldsymbol{G}$. In short: each component of $\boldsymbol{G}$ must contain some path from the initial state back to itself, via which all initially-constrained coordinates are increased. The reverse property must also hold: If the component is reversed (all transitions (q, v, q') become (q', v, q)) there must be a path from the final state to itself along which all finally-constrained coordinates are strictly *decreased*.

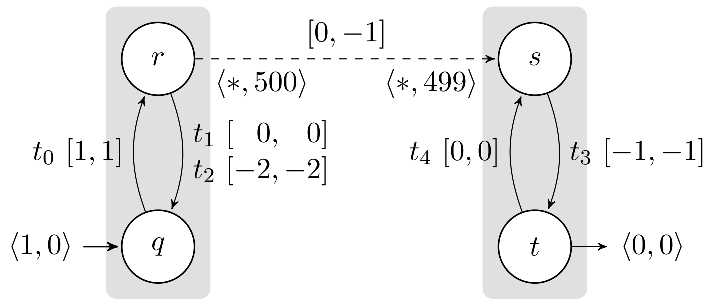


Figure 5.5: A two-component GVASS whose first component satisfies θ_2 and whose second component does not.

Finding a path on which all coordinates are increased evidences some amount of "pumpability": we can repeat this path in order to pump the values of all unconstrained coordinates arbitarily high.

This is subtly different in a number of ways to the stipulation made in θ_1 that a pseudo-run must exist along which unconstrained coordinates can become unboundedly large. Firstly, the path satisfying θ_2 for some component must form a (part of a) legal VASS run, and not just a pseudo-run—that is, the coordinates must not drop below zero on the path. Secondly, we require that the values are increased by the end of this path, and not only at some point along it. Finally, all unconstrained coordinates must be increased at the same time—it is not sufficient that each such coordinate is unbounded on *some* path.

Observe that there are a couple of common situations in which θ_2 self-evidently holds. For example, a component comprising one state q with the initial and final coordinates all constrained will pass θ_2, since there is a zero-length path from q to q on which the value of all unconstrained coordinates (vacuously) increase. Hence θ_2 is not meaningful for the examples in Figures 5.3 and 5.4. Figure 5.5 includes a more interesting example in which θ_2 holds for the first component but not the second.

5.2.2 Cleaning GVASSs

Throughout the algorithm, a number of assumptions are made about the shape of GVASS components. We must preprocess ("clean") any given GVASS to ensure that these assumptions hold. After performing any decomposition, it is possible that the decomposed GVASSs will once again violate the assumptions; so these cleaning steps must be repeated after every decomposition.

Strong Connectedness

It is assumed that every component in every GVASS is strongly connected. To accomplish this we must decompose the GVASS whenever we find a non-strongly-connected component.

Definition 5.6. The *state-transition graph* of a GVASS component $\mathcal{C}$ is a directed (possibly multi)graph whose vertices are labelled by the states of $\mathcal{C}$ and whose edges are (q, r) for each (q, v, r) in the set of transitions of $\mathcal{C}$.

Definition 5.7. A GVASS component $\mathcal{C}$ is *connected* if and only if the vertex labelled by the final state of $\mathcal{C}$ is reachable from the vertex labelled by the initial state of $\mathcal{C}$ in the state-transition graph of $\mathcal{C}$. $\mathcal{C}$ is *strongly connected* if and only if every vertex is reachable from every other vertex in the state-transition graph of $\mathcal{C}$.

In our diagrams, the state-transition graph for a component is found by removing the vectors attached to each transition and ignoring the initial/final constraints. Unusually for Kosaraju's algorithm, the strong connectedness property can be determined efficiently—the strongly connected subsets of states for each component can be determined in time linear in the number of states and transitions. (We shall discuss how in Section 5.3).

When evaluating strong connectedness, we may learn one of three outcomes. If the component is not connected, there can never be a GVASS run through the component. Since a GVASS run must traverse *every* component, there can never be such a run and hence we may immediately deduce that the target is unreachable. (The GVASS decomposition process will never result in a disconnected component, so if a component is disconnected then the original VASS must also have been disconnected.) If the component is strongly connected, it requires no further cleaning and we may proceed. If it is connected but not strongly connected—if there is some vertex in the state-transition graph of the component that is not reachable from some other—then we must perform a particular type of decomposition. This is slightly different to the decompositions described in Section 5.2.3, as it is not derived from violations of the θ condition.

If the component is not strongly connected, then we are able to partition the states of the component into $n > 1$ strongly-connected subgraphs (SCSs)[3] such that the edges between any two SCSs X and X' either all go from some state in X to some state in X' or vice versa. (If they went both ways, the two would form one larger SCS.) The decomposition of the component consists of

[3]In graph theory, the internally strongly connected subgraphs of a graph are called *components* (SCCs). To avoid confusion with the GVASS notion of components we shall stick to subgraphs (SCSs).

characterising the possible paths from the SCS containing the initial state to the SCS containing the final state, in terms of the edges between these SCSs.

Example 5.3. Consider the GVASS in Figure 5.6, which consists of one connected but not strongly connected component. The vectors on transitions are arbitrary.

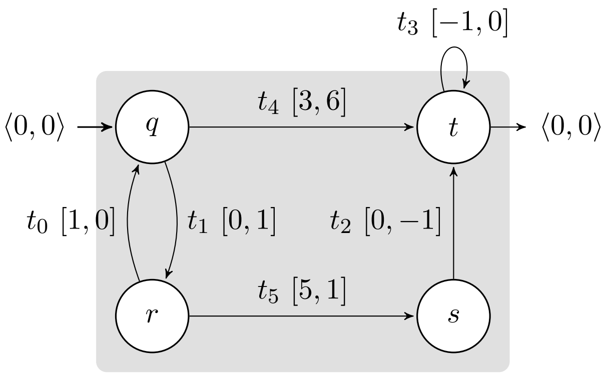


Figure 5.6: A one-component GVASS which requires SCS decomposition.

Strongly-connected subgraph analysis shows that there are three strongly connected subgraphs, namely $\{q, r\}$, $\{s\}$, and $\{t\}$. The transitions t_2, t_4 and t_5 label the edges between the subgraphs. Hence the analysis shows us that any path passing through this component may fire any one of those transitions at most once. More precisely we can characterise every path through this component as being of one of the two shapes given in Figure 5.7.

$$\begin{tikzcd}
\langle 0,0\rangle \arrow[r] & \{q,r\} \arrow[loop above, "\{t_0, t_1\}"] \arrow[r, "t_4"] & \{s\} \arrow[r, "t_2"] & \{t\} \arrow[loop above, "\{t_3\}"] \arrow[r] & \langle 0,0\rangle
\end{tikzcd}$$

$$\begin{tikzcd}
\langle 0,0\rangle \arrow[r] & \{q,r\} \arrow[loop above, "\{t_0, t_1\}"] \arrow[r, "t_5"] & \{t\} \arrow[loop above, "\{t_3\}"] \arrow[r] & \langle 0,0\rangle
\end{tikzcd}$$

Figure 5.7: Characterisations of paths through the component of the GVASS in Figure 5.6 as determined by SCS decomposition.

(Note that unreachable states will automatically be removed by this process, since they will not appear in any of the characterisations of runs through the component.) The decomposition of a component is dictated by the characterisation of all possible paths through strongly-connected subgraphs. Each such path gives rise to a new GVASS, where:

- each strongly-connected subgraph becomes its own component, compris-

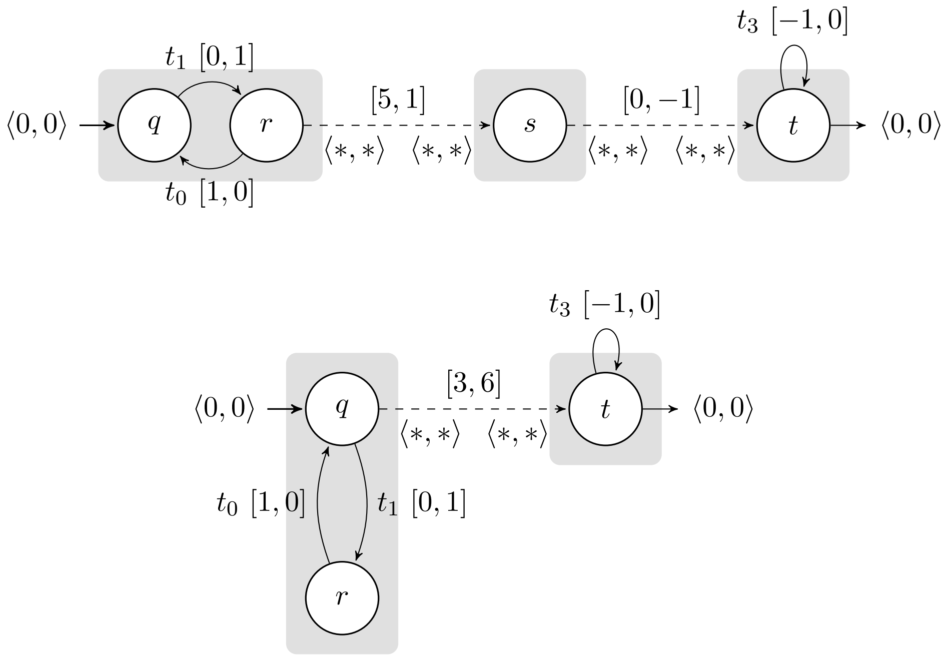


Figure 5.8: The two decompositions of the GVASS in Figure 5.6 as determined by strongly connected subgraph decomposition.

ing only the states and transitions labelling nodes and edges internal to that subgraph;

- the edges between adjacent subgraphs become the adjoinments between the two corresponding components, with the vector of that adjoinment being the vector of the transition labelling that edge; and
- all coordinates are unconstrained, except for the initial coordinates of the first component and the final coordinates of the last component, which have the initial and final values (respectively) of the original component.

Continuing Example 5.3, the two shapes of path in Figure 5.7 gives rise to the two GVASSs in Figure 5.8. Each of the newly decomposed GVASSs satisfies the strong connectedness requirement in each component. It is notable that the number of such shapes of path may be exponential in the size of the component, and so performing this strongly-connected decomposition only once may result in exponentially many GVASSs, each of which might need decomposing further. Hence the complexity implications of this intermediate step alone are alarming.

Trivial component elimination

A component is called *trivial* if it has zero transitions. In such cases, we can remove the component entirely and unify the inbound and outbound adjoin-

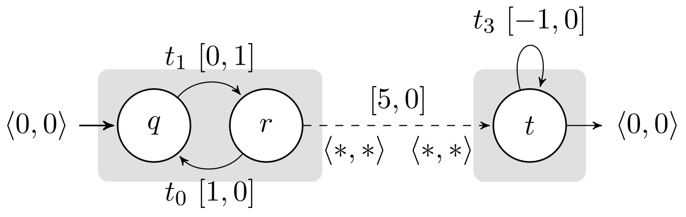


Figure 5.9: The first GVASS from Figure 5.8 after undergoing trivial component elimination.

ments and constraints.

The first decomposition in Figure 5.8 contains a trivial component with state s. We replace it by adjoining the components before and after the trivial component, with a vector equal to the sum of the inbound and outbound vectors. If there are any constraints on the initial/final coordinates of the trivial component, we unify those constraints with the initial/final constraints of the adjacent components. (If the constraints being moved are incompatible with those already present at the adjacent components, then no run can exist, and so we discard the GVASS.) Figure 5.9 shows the effect of eliminating the trivial component with state s.

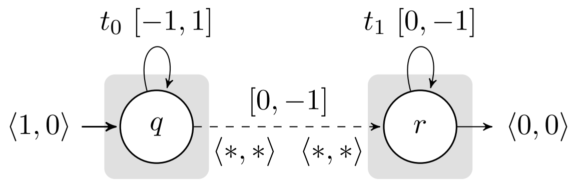
5.2.3 θ-driven Decomposition

We call the graph derived from taking every possible (iterated) decomposition of a graph $\boldsymbol{G}$ the *decomposition tree* rooted at $\boldsymbol{G}$. We shall see in Section 5.2.4 that this is indeed acyclic and that it is guaranteed to be finite. Each of the decompositions that is not the result of cleaning is derived from exactly one violation of the θ condition. Each type of violation corresponds to a particular flavour of decomposition. We shall go through each in turn with minimal representative examples.

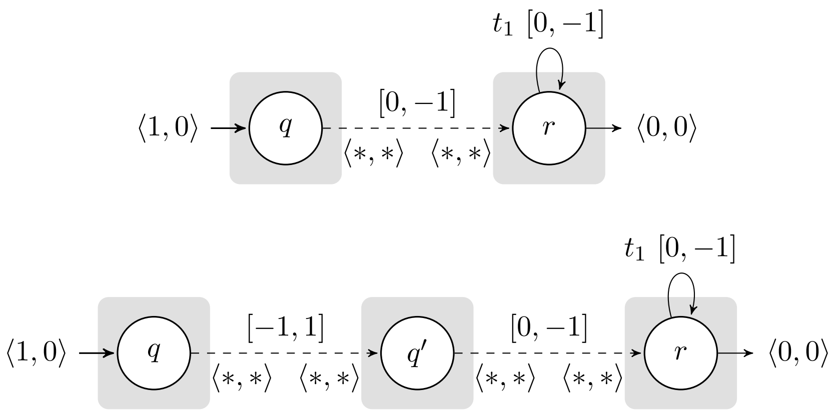
It is possible that a GVASS violates both θ_1 and θ_2, and/or that it violates them in more than one way at the same time. If this is the case, we expand on only one of the violations (which may itself result in several children, as seen shortly). This guarantees that we do not inadvertently rule out lawful runs by failing to consider interleavings.

Violating θ_1

Recall that θ_1 mandates that a pseudo-run exists in $\boldsymbol{G}$ on which we can use every transition an unbounded number of times, and every unconstrained coordinate can reach unboundedly high values. Hence, if θ_1 is violated, we know that some transition may only be used a bounded number of times, or some



(a) In this GVASS, transition t may be fired at most once.



(b) The two GVASSs derived from t_0's violation of θ_1, one with zero firings of t_0 (above) and one with one (below). After decomposition, both q and q' would be removed by cleaning as they are now trivial.

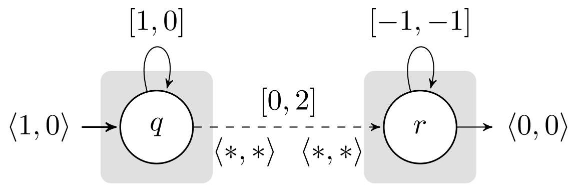
Figure 5.10: A simple GVASS before and after decomposition by removing a transition violating θ_1.

unconstrained coordinate has a maximum value that can be attained in some component, on any pseudo-run. (If this is true for all pseudo-runs then it is true for all runs, since pseudo-runs are a relaxation of runs.)

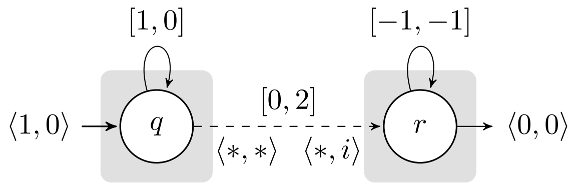
Transition t is bounded. Suppose we learn that transition t in component $\mathcal{C}$ in GVASS $\boldsymbol{G}$ can only be fired at most n times. If so, we decompose creating $n+1$ copies of $\boldsymbol{G}$, where in each copy $\boldsymbol{G}_i$ (for $i \in \{0,\ldots,n\}$) the component $\mathcal{C}$ is replaced by i copies of $\mathcal{C}$, each adjoined by the vector of t and with all non-rigid coordinates unconstrained between copies. We are then able to remove t from all copies.

Intuitively, each decomposition $\boldsymbol{G}_i$ represents the universe in which we fire t exactly i times. If a run exists, then it must exist in one of those universes. Figure 5.10 gives an example of a GVASS ($\boldsymbol{G}_{\mathsf{ex}}$ from Example 5.2, after cleaning) in which transition t_0 may only be fired at most once; and the result after applying this decomposition.

Coordinate x is bounded. If we learn that unconstrained initial or final coordinate x may not exceed n while (pseudo-)running through component $\mathcal{C}$, then we may constrain x. We generate $n+1$ copies, $\boldsymbol{G}_0$ through $\boldsymbol{G}_n$, and in



(a) In this GVASS, it can be determined that coordinate 2 will never exceed 2 when entering the second component.



(b) The GVASS from (a) decomposes into three GVASSs, $\boldsymbol{G}_0$, $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$, where the initial value of coordinate 2 in the second component is constrained to i for each $\boldsymbol{G}_i$.

Figure 5.11: A simple GVASS before and after decomposition by removing a transition violating θ_1.

each $\boldsymbol{G}_i$ the value of x is initially constrained to i in $\mathcal{C}$.

By constraining the intial value of x we are placing a constraint on the transitions earlier in the GVASS which increase the value of coordinate x, which might then result in more θ_1 based decompositions in future rounds. In Figure 5.11a, we can learn that coordinate 2 may not exceed 2 initially in the second component; the result of decomposition is three copies, each constraining coordinate 2 to a different value from $\{0, 1, 2\}$.

Violating θ_2

For the sake of simplicity, we shall work on the assumption that the violation of θ_2 is found in the forward version of the condition. If the value was found to be bounded in the backward version, the process is the same, swapping “initial” and “final”.

Suppose that θ_2 is violated for coordinate x in component $\mathcal{C}$. If so, the value of coordinate x must be bounded everywhere within the state space of $\mathcal{C}$. We can learn from the violation the maximum value that coordinate x can attain; call it n.

The process by which we decompose according to a violation of θ_2 depends on whether the coordinate in question is finally constrained or unconstrained. If it is finally unconstrained, then we decompose by constraining component n to all of $\{0, \dots, n\}$ (in a similar way to the constraints imposed for a bounded coordinate in θ_1).

If it is already finally constrained, then we now know an initial and final constraint for the coordinate and (possibly an overapproximation of) the complete range of values it may take in the component. Therefore, within $\mathcal{C}$ the value of coordinate x no longer acts as a counter; instead we can treat it like a finite piece of state. We do this by encoding the possible values of coordinate x directly into the states of the component by a product construction. The coordinate x is then marked as rigid, with the rigid value set to the value to which it was previously initially constrained. If the initial and final constrained values were different, then when making the coordinate rigid that delta is no longer part of the final constrained vector. We account for this by modifying the adjoinment which follows the component by adding $v'[i] - v[i]$ to the ith component of the adjoinment.

As one may imagine, the result of applying this product construction to states and transitions results in massive blowup even for relatively small components. Thankfully, it is unusual in practice to see θ_2 violated without θ_1 having been violated first. Since only one violation is considered at a time, and our implementation checks θ_1 before θ_2, the problematic components have usually been decomposed before their θ_2 violations would be considered.

5.2.4 Termination

Firstly, note that termination of the reachability procedure is equivalent to finiteness of the decomposition tree rooted at $\boldsymbol{G}$. By König's Lemma, we need only show that the decomposition tree has finite branching everywhere and finite depth. Finite branching is easy: each of the decompositions (from cleaning and from θ-violations) generates only a finite number of decompositions. Finite depth is guaranteed by an ingenious encoding of the size of a GVASS.

Definition 5.8. The *size* $|\mathcal{C}_i|$ of a component

$$\mathcal{C}_i = (d, Q_i, T_i, q_i, q_i', R_i, r_i, C_i, U_i, C_i', U_i', v_i, v_i')$$

is the triple

$$(d - |R_i|, |T_i|, |U_i + U_i'|).$$

The size $|\boldsymbol{G}|$ of a GVASS $\boldsymbol{G}$ is the multiset of the sizes of the components of $\boldsymbol{G}$.

Let us define also a refining relation for GVASS sizes. We say that $|\boldsymbol{G}'|$ *refines* $|\boldsymbol{G}|$ if and only if one can attain $|\boldsymbol{G}'|$ by removing (one copy of) some triple from $|\boldsymbol{G}|$ and adding (a finite number of) lexicographically smaller triples. Take $\preccurlyeq$ to be the reflexive, transitive closure of the refining relation. Thanks to Dershowitz and Manna [34], since the size of a GVASS is a multiset over lexicographically-ordered triples of natural numbers (themselves forming a well-quasi-ordering), we know that $\preccurlyeq$ is a well-quasi-ordering.

Remark 5.1 ([90, from Theorem 7]). *If $\boldsymbol{G}'$ can be attained by applying a single decomposition to $\boldsymbol{G}$, then $|\boldsymbol{G}'|$ refines $|\boldsymbol{G}|$.*

The above remark can be shown by checking each of the different flavours of decomposition. In every decomposition, exactly one component $\mathcal{C}_x$ is replaced with a family of chains of zero or more components $\mathcal{C}_i$ such that in every $\mathcal{C}_i$:

- The number of rigid components is larger than in $\mathcal{C}_x$;
- The number of transitions has been reduced by 1; or
- The number of initially or finally unconstrained coordinates has decreased.

In the cases where a decomposition replaces a component by more than one component (for example, when a transition is found by θ_1 to be bounded), some new components $\mathcal{C}_i$ may have more unconstrained coordinates than $\mathcal{C}_x$. In these cases, the number of transitions has been reduced, and so $|\mathcal{C}_i| < |\mathcal{C}_x|$ is maintained. Likewise, the product construction of the θ_2 decomposition will massively increase the number of transitions in a component; but in doing so the number of rigid components is increased and so the first component of the size decreases, so maintaining $<$ for the component's size and hence the decomposition still results in a refinement of the GVASS's size.

Since $\preccurlyeq$ forms a well-quasi-ordering, and a decomposition of a GVASS $\boldsymbol{G}$ is always strictly smaller than $\boldsymbol{G}$ under $\preccurlyeq$, we can infer that any sequence of decompositions must necessarily be finite. Hence the decomposition tree rooted at any GVASS will only contain paths of finite length and so, by König's Lemma, the tree must be finite. Since Kosaraju's algorithm performs an exhaustive search of the decomposition tree, we have that the procedure will terminate.

5.2.5 Correctness

We shall give an informal outline of a correctness argument for Kosaraju's algorithm. The intuition behind the use of θ_1 and θ_2 is as follows: If both θ_1 and θ_2 are satisfied, then a run can be constructed that pumps unconstrained coordinates arbitrarily high (via θ_2); then performs a pseudo-run (as dictated by θ_1); and finally pumps the unconstrained values back down (by the latter part of θ_2). If the unconstrained values are pumped sufficiently high, then any pseudo-run satisfying θ_1 will become a run. Furthermore, if such a pseudo-run exists, then one can be chosen such that the concatenation of pumping-up, following the pseudo-run, and then pumping down, is guaranteed to reach the exact value required of the final vector.

An alternative way of formulating the relationship between the two conditions is that θ_1 identifies pseudo-runs, while θ_2 ensures that, via pumping, at least one such pseudo-run will be able to operate entirely within the non-negative integers. Note that the "pumping-up" need not happen in the very first component of a GVASS; there may be some constrained part before an unbounded part of a run. The converse is true about the "pumping-down"; and in fact both of these processes may occur multiple times in the same run.

Any time values become constrained the problem becomes easier. If all values eventually become constrained, θ_2 trivially holds and the problem can be decided entirely through integer linear programming—either a solution will exist or it will not, and so θ_1 will hold (and the target is reachable) or no valid decompositions will exist (and the target is unreachable).

A concise and well-reasoned argument for correctness of Kosaraju's algorithm is due to Lasota [96], which proceeds firstly by formulating the θ condition over standard VASS before extending it to GVASS.

5.3 `KReach`: Implementing Kosaraju's Algorithm

We have implemented Kosaraju's algorithm in a tool, `KReach`, which can be used to decide instances of the general Petri net reachability problem. In the same spirit as Chapter 4, we shall now present a sampling platter of implementation details relating to `KReach` in order to give an insight into the approaches and challenges involved.

The code segments given in this section are derived from the source code of `KReach`, which is available online at GitHub[4]. The code is written in the Haskell programming language.

The algorithm is represented as a function `kosaraju` which initially takes as input a list of GVASSs, and returns a `KResult`: either `KHolds` or `KDoesNotHold`, indicating a result of reachable or unreachable repsectively. Rather than explicitly building and searching the decomposition tree, we explore the tree by recursively calling `kosaraju` on the list of descendants if the θ condition does not hold; only when all children have returned a result of `KDoesNotHold` do we finally return `KDoesNotHold` overall. By contrast, if any such child returns `KHolds`, we immediately return `KHolds` and terminate the

[4]`https://github.com/dixonary/kosaraju`

```
kosaraju
  :: [GVASS]                       -- A list of input GVASSs
  -> IO KResult                    -- The result, via some IO
kosaraju vs = do
  let vs' = concatMap clean vs -- Clean every GVASS
  let decomps = map checkGVASS vs'
  if
    -- Search space exhausted
    | decomps == []           -> return KDoesNotHold
    -- Answer found
    | Nothing `elem` decomps -> return KHolds
    -- More GVASSs to check
    | otherwise               -> kosaraju (takeJusts decomps)



checkGVass
  :: GVASS                         -- Take one GVASS as input
  -> IO (Maybe [GVASS])            -- Return refinements or Nothing
checkGVASS gvass = do
  case ( θ₁ gvass, θ₂ gvass) of
    -- If both θ₁ and θ₂ hold, we are done
    (ThetaOneHolds, ThetaTwoHolds) -> Nothing

    -- θ₁ is violated ( θ₂ is irrelevant)
    (thetaOneViol , _             ) -> do
      decomps <- refine θ₁ gvass thetaOneViol
      return (Just decomps)

    -- θ₁ holds but θ₂ is violated
    (ThetaOneHolds, thetaTwoViol ) -> do
      decomps <- refine θ₂ gvass thetaTwoViol
      return (Just decomps)
```

Figure 5.12: The main loop of the `kosaraju` function and the `checkGVASS` function on which it relies.

program.

Figure 5.12 shows the fundamental structure of the main `kosraju` loop. Note that, as with `HCover`, we make use of the `IO` context to interface with the outside world. In particular we use it for logging intermediate values such as the instance size and for communicating with an SMT solver. The structure of the loop is a breadth-first search rather than a depth-first search; all the instances at the same "level" of the decomposition tree are checked in succession. For unreachable instances there is no real difference in approach; for reachable instances it is conjectured that a breadth-first strategy will reach an answer of `KHolds` more quickly (because, in short, every θ-driven decomposition produces a family of "universes" of runs, and only one of those is likely to result in a real run).

5.3.1 Cleaning

Firstly the strongly-connected subgraph decomposition is applied to every component in every `GVASS` at each step. For each component this may generate a family of chains of components. In order to minimise overhead, each component is decomposed in this way simultaneously and the Cartesian product of these chains is taken. Recalling the terminology of Section 5.2.2, if we have a GVASS of three components, where SCS decomposing the first component results in three shapes of runs, the second component two shapes of runs, and the third results in four shapes of runs, the complete subgraph decomposition results in $2 \times 3 \times 4 = 24$ total GVASSs.

In `KReach`, the SCS decomposition is implemented by calling to the `Data.Graph` module which forms part of the core `containers` library. This module includes a first-class notion of strongly-connected subsets[5] and so using it to compute the subsets is relatively simple. Deriving the state-transition graph from a given component, and inferring the new components based on the result of the computation, is more complex but ultimately unproblematic.

5.3.2 Checking θ_1

When considering pseudo-runs rather than runs, we eliminate the requirement that the values of coordinates must be nonnegative. In doing so we remove most constraints on the order in which transitions can be fired in the machine. Those that remain can be encoded into first-order logic over $\mathbb{Z}$ and handed off to an SMT solver.

The state equation. The first constraint that must be in place when checking θ_1 is the so-called *state equation* for vector addition systems. The state equation mandates that the final value in any coordinate is equal to the initial value in that coordinate plus the sum of the changes undergone by firing of transitions. To wit, for initial vector v, final vector v' and run $\sigma = (t_1, \ldots, t_\ell)$:

$$\forall i \in \{1, \ldots, d\} : v'[i] = v[i] + \sum_{x \in \{1,\ldots,\ell\}} t_x[i].$$

Since the order of transitions is irrelevant, in our encoding we only care about the number of times each transition fires. This is equivalent to working with the Parikh image π_σ of the run (see Definition 3.12), so we may rephrase the

[5]`https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Graph.html#v:stronglyConnComp`

state equation as

$$\forall i\{1,\ldots,d\} : v'[i] = v[i] + \sum_{t\in T} \pi_\sigma(t)\cdot t[i].$$

Kirchoff constraints. Secondly we need to enforce the so-called *Kirchoff constraints* on transitions. Kirchoff's first law states that the sum of the flow into a (non source or sink) node must be equal to the sum of the flow out of a node. This phrasing is general as it applies to graph theoretic problems but also in other fields including hydrodynamic and electronic engineering. The same law must hold in our setting. Consider any walk through the state-transition graph of a component, starting at the vertex labelled by the initial state and ending at vertex labelled by the final state. The number of times each vertex is visited must be equal to the sum of the times in-edges are traversed to that vertex; and also equal to the sum of the times out-edges are traversed from that vertex. (At the initial and final vertices, we expect the out-degree and in-degree to be 1 higher, respectively).

Inherited GVASS constraints. We also impose the constraints derived from the metadata on GVASS components, namely known initial and final values of constrained or rigid coordinates in each component, and relationships between final and initial values of adjacent coordinates derived from an adjoinment.

To encode the above properties into first-order logic, we take the following set of variables:

- One variable for each of the initial value $I_{\mathcal{C},i}$ and final value $F_{\mathcal{C},i}$ of every coordinate i in every component $\mathcal{C}$.
- One variable K_t for the number of times each transition t is fired in the run. (A disambiguating schema is used to avoid naming collisions where transitions appear in multiple components; we will assume disjointness of names here.)

This produces the set of sentences given in Figure 5.13. The conjunction of all of them forms the integer linear programming problem given to the solver. The universal quantifications and summations are fully expanded in the version given to the solver, and each sentence is rearranged to leave a sum of some values on the left and a zero on the right.

The solutions to the integer linear programming problem take the form of a semilinear set (B, P) where B is the set of bases and P the set of periods. If an unconstrained coordinate has a nonzero value in any period, then there must

$$
\begin{aligned}
F_{\mathcal{C},i} &= I_{\mathcal{C},i} + K_t \cdot t[i] && \forall\, \mathcal{C}_x \in \boldsymbol{C}, i \in \{1,\ldots,d\}, t \in T_x && (1)\\
\Sigma_{t \in \mathrm{out}(s)} K_t &= \Sigma_{t \in \mathrm{in}(s)} K_t && \forall\, \mathcal{C}_x \in \boldsymbol{C}, s \in Q_x \setminus \{q_x, q'_x\} && (2)\\
\Sigma_{t \in \mathrm{out}(q_x)} K_t &= \Sigma_{t \in \mathrm{in}(q_x)} K_t - 1 && \forall\, \mathcal{C}_x \in \boldsymbol{C} && (3)\\
\Sigma_{t \in \mathrm{out}(q'_x)} K_t &= \Sigma_{t \in \mathrm{in}(q'_x)} K_t + 1 && \forall\, \mathcal{C}_x \in \boldsymbol{C} && (4)\\
I_{\mathcal{C}_x,i} &= v_x[i] && \forall\, \mathcal{C}_x \in \boldsymbol{C}, i \in C_x && (5)\\
F_{\mathcal{C}_x,i} &= v'_x[i] && \forall\, \mathcal{C}_x \in \boldsymbol{C}, i \in C'_x && (6)\\
I_{\mathcal{C}_x,i} &= r_x[i] && \forall\, \mathcal{C}_x \in \boldsymbol{C}, i \in R_x && (7)\\
F_{\mathcal{C}_x,i} &= r_x[i] && \forall\, \mathcal{C}_x \in \boldsymbol{C}, i \in R_x && (8)\\
I_{\mathcal{C}_x,i} &= F_{\mathcal{C}_{x+1},i} + z[i] && \forall\, \mathcal{C}_x, \mathcal{C}_y \in \boldsymbol{C}, (q'_x, z, q_y) = a_x, i \in \{1,\ldots,d\} && (9)
\end{aligned}
$$

Figure 5.13: The first-order logic sentences that comprise the ILP problem describing θ_1. The sentences encode the Parikh image constraints (1), Kirchoff constraints (2-4), initial and final values for constrained (5-6) and rigid (7-8) values, and adjoinment relationships (9).

be some way in which that coordinate can reach unboundedly high. Conversely, if there is *no* period in which the coordinate is nonzero, then there is an upper bound on the height that value can reach—and moreover that upper bound is the maximum value in B. The same principle applies for transitions. We return the first violation found this way and decompose accordingly. If the ILP problem returns no solutions at all, we know that θ_1 will *never* hold, and thus we return that we should give up on the GVASS. If the ILP problem has solutions *and* all transitions and unconstrained coordinates have nonzero periods, then we can conclude the existence of a pseudo-run satisfying θ_1 and we report that `ThetaOneHolds`. Decompositions are as described in Section 5.2.3.

5.3.3 Checking θ_2

Recall from Section 4.3.1 the Karp-Miller procedure for computing the coverability tree of a VASS from an initial configuration. θ_2 requires that we find some run by which we can get from the initial state q of our component back to q, and in doing so strictly increase the values in all constrained coordinates. For now let us consider only the forward direction.

We can formulate the condition in the following way. For a component $\mathcal{C}_i = (d, Q_i, \ldots)$, is there some run from (q_i, v) to (q_i, v^+) for some $v^+ \geq v + \vec{1}$, in a restricted version of the VASS (Q_i, T_i) where we project to the coordinates of C_i? Astute readers will recognise this as a coverability problem, and indeed we solve this by reducing to coverability.

However, it is not sufficient only to know whether (q_i, v^+) is coverable or not—we must produce a violation which can be acted upon to decompose the

GVASS. If (q_i, v^+) is not coverable, we can infer that at least one coordinate x must be bounded everywhere. To learn the values that coordinate x can take, we must look through the whole space of coverable vectors. To do this analysis we produce and look through the complete Karp-Miller tree for the problem. Coverability of (q_i, v^+) is equivalent to the presence of $(q_i, (\omega, \dots, \omega))$ in the tree. If it is absent, we return the violation $(x, m(x))$ where $m(c)$ is the maximum value of coordinate c on any node in the Karp-Miller tree and x is the smallest coordinate for which $m(x) \neq \omega$.

To this end, the suite of libraries included as part of the release of KReach includes karp-miller[6], a complete implementation of the Karp-Miller procedure for determining the coverability tree of a VASS starting from a given initial configuration. As a separate library, karp-miller does not rely on any of the logic internal to kosaraju and can easily be repurposed for other procedures. The high-level algorithm[7] comprises less than 40 readable lines of well-commented, type-directed Haskell code, which hopefully make for edifying reading. The same principles and best practices have been followed in karp-miller as in kosaraju—it too may serve as a useful learning tool and/or platform for future improvements such as those of Finkel, Haddad and Khmelnitsky [54]. The potential exists to spin karp-miller out into a coverability checker in its own right but this has not yet been done.

5.3.4 Optimisations

In spite of the reprehensible algorithmic complexity of the reachability problem, some effort was still undertaken to show that the algorithm can be optimised. A number of small improvements have been implemented which do not change the character of the algorithm.

Firstly, consider the decompositions produced by a violation of θ_1. Say that we learn that the number of firings of some t is bounded. This is reflected in the solution to the integer linear programming problem: the values of K_t in the periods of the semilinear set will all be zero. Kosaraju's original algorithm (as explained in Section 5.2.3) tells us to generate the family of decompositions where t is fired $0, 1, \dots, n$ times where n is the maximum value of K_t in any basis of the semilinear set. However, we can be more precise: the solution is telling us that the transition may be fired only *exactly* as many times as the values of K_t in the bases. So, for example, if the basis values for K_t are 0, 3 and 10, we need only generate those chains corresponding to 0, 3 and 10 firings of t. The same principle applies when constraining coordinates initially

[6]https://github.com/dixonary/karp-miller

[7]https://github.com/dixonary/karp-miller/blob/cf24259/src/Data/VASS/Coverability/KarpMiller.hs#L46

and finally.

In addition, searching the decomposition tree breadth-first instead of depth-first allows us to parallelise computation in each level of the tree. We make use of Haskell's lightweight concurrency primitives to parallelise the (computationally intensive) checking of θ for each GVASS at a given level; the result across all GVASSs (KHolds, KDoesNotHold, or further decompositions) is computed once those checks are complete.

5.4 Experimental Results

Given that KReach is believed to be (at time of publication) the first complete implementation of a decider for the general VASS reachability problem, it is unsurprising that benchmarks and sample tests are difficult to come by. The experimental results presented in this section are a mixture of synthetic nets produced to superficially investigate properties of the procedure, and instances derived from coverability.

5.4.1 Testing KReach on synthetic instances

A simple synthetic example was constructed to investigate how small changes in problem instances can affect the running time of Kosaraju's algorithm. We parameterise the problem instance V_{ex} from Figure 5.2b by X: X is the initial value of the first coordinate and $-X$ is the value of the second coordinate on the adjoinment between states a and b. For each such instance there is exactly one run that reaches the final configuration (X firings of t_0 followed by one firing of t_1). The diagram of the lifted GVASS $\boldsymbol{G}(V_{\text{ex}}(X), (X, 0), (0, 0))$ is given in Figure 5.14.

As noted for HCover, the library used to implement integer linear programming in Haskell (SBV) is agnostic towards which solver is used on the system. However, different solvers have different properties which may be exploited by certain shapes of problem. Two leading solvers, z3 and cvc4, were tested against our synthetic problem instance for values of X up to 10. Figure 5.15 shows the relative performance of z3 against cvc4. This preliminary test suggests that cvc4 far outperforms z3 on all but the smallest instances. The time taken in z3 grows exponentially whereas in cvc4 it is only slightly superlinear (this is closer to what is expected; after one round of decomposition, the number of variables in the problem instance will be linear in X). Observe also that there is a reproducible *drop* in running time between $X = 4$ and $X = 5$ for z3; we conjecture that there is some threshold at which z3 changes the heuristics it uses, and that threshold is met for our instance when

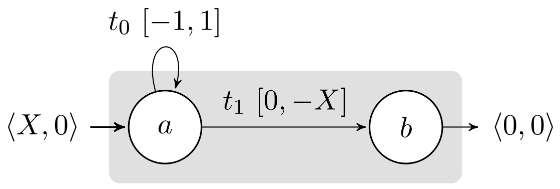


Figure 5.14: Our original sample case $\boldsymbol{G}_{\mathrm{ex}}$, parameterized over X.

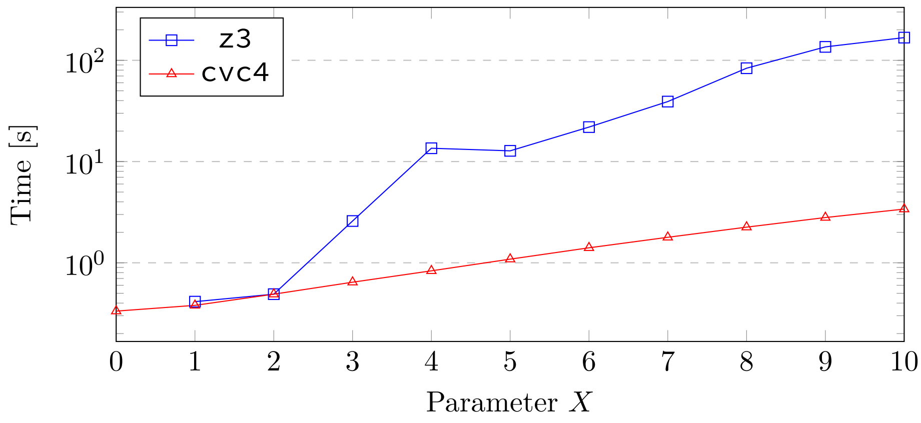


Figure 5.15: Time against parameter X for $\boldsymbol{G}(V_{\mathrm{ex}}(X), (0, X), (0, 0))$ with the supported solvers. Shown is the average time taken across 10 runs for each X and for each solver.

X exceeds 4.

5.4.2 Testing `KCover` on real instances

Recall from Proposition 5.1 that instances of the coverability problem can be reduced to instances of the reachability problem. Hence, one way that we can test `KReach` is to ask it to solve coverability instances. A sister tool to `KReach`, called `KCover`, is included within the `KReach` binary; in order to run `KCover` one need only add the `-c` flag when invoking `KReach` from the command line.

The procedure implemented by `KCover` is exactly that described in the proof of Proposition 5.1: given some coverability query, convert the final vector into a transition which leads to a distinguished state ∇, in which all coordinates can be wound down to zero; and then use `KReach` to check for reachability of $\vec{0}$. The overheads of converting from a coverability query to a reachability query are minimal.

`KCover` allows us to use benchmarks for the coverability problem as a source of test cases for the reachability algorithm. The suite of benchmarks provided with the tool includes a number of test cases for various aspects of the implementation, as well as examples from the non-elementary lower bound

Instance	Outcome	MIST (s)	QCover (s)	ICover (s)	KReach (s)
Kanban	safe	404	TLE	TLE	1
Bingham_h150	safe	TLE	TLE	TLE	533
Manufacturing	safe	1	0	0	4
Bug_Tracking_x0	safe	MLE	13	33	TLE
PNCSACover	unsafe	3	27	59	TLE

Table 5.1: Sample of test cases. MLE = Memory Limit Exceeded; TLE = Time Limit Exceeded.

construction of Czerwiński et al. [30].

KCover was evaluated against a host of problems and a number of solvers from the existing literature on coverability checkers. To quickly recapitulate from Chapter 4: QCover [13] implements coverability based on relaxation to continuous coverability; ICover [63] refines this further with inductive invariants; MIST [58–60] implements backwards coverability search with a range of pruning criteria and an efficient representation of upward-closed sets.

Table 5.1 includes some specific instances which are representative of the broader trends in experimental results. The results shown were computed on an early-2017 MacBook Pro with a 2.3GHz dual-core Intel Core i5 CPU and 8GB of LPDDR3 RAM. The time limit was set to one hour, with a memory limit of 4GB RAM. On many safe cases, such as Kanban and Bingham, KCover is able to determine safety faster than state of the art coverability solvers by finding zero valid refinements (terminating the search immediately). On some safe nets such as Manufacturing, KCover cannot immediately rule out coverability in this way, and the refinement tree must be explored. The Bug_Tracking examples induced intractably large integer linear programming problems. Unsafe cases such as PNCSACover induced large refinement trees, which were unable to be explored fully within the time limit.

5.5 Concluding Remarks and Further Work

In addition to the karp-miller library mentioned previously and the KReach tool itself, a library called vass[8] has been created to implement a modular and extensible interface for working with Vector Addition Systems with States in the Haskell langauge.

The experimental results above are of particular value for further study, especially for the uncoverable instances. That Kosaraju's algorithm is sometimes able to rule out coverability quickly implies that it may be a fruitful source of static invariants for some classes of Petri nets. One line of further

[8] https://github.com/dixonary/vass

work may be to attempt to formally classify those nets for which Kosaraju's algorithm is effective in practice.

At present `KReach/KCover` only implements one parser which is for the `MIST .spec` problem specification format. (When reading such files as reachability instances, it replaces $\geq$ in the specification of the target vector by $=$). There are other formats that describe coverability problems. For example, the Petri Net Markup Language format `.pnml`[9] is an XML-based interchange format which is generalisable over different types of Petri nets, and is now an ISO standard. By implementing a parser for this or other interchange formats it may become possible to use `KCover` on a wider variety of problem instances.

We noted in passing that `KReach` uses a breadth-first search strategy for searching the decomposition tree, and that it is conjectured that this results in termination more quickly than a depth-first strategy. Determining empirically whether a breadth-first or depth-first search is more effective in practice may make for potential future work. Taking inspiration from the recent works of Amat, Dal Zilio and Hujsa [6] and Blondin, Haase and Offtermatt [15], it may be possible to introduce clever heuristics for exploring the tree based on which decompositions are most likely to result in a solution.

There is also plenty of scope for improving the performance of `KReach`. For example, there is scope to adapt `KReach` in light of the theoretical improvements over the original 1982 formulation, not least those of Lambert [95]. Further, there may be optimisations that can be derived from the novel theoretical developments in the Ackermannian upper bound proof of Leroux and Schmitz [104]. Finally, there is scope to build some parsers and problem translators which allow for evaluating instances of problems that are known to reduce to reachability in Petri nets, for example in logic [33, 86], concurrent systems [48], or process calculi [111].

[9]`https://pnml.org`

Chapter 6

Leafy Automata

Chapter 5 concludes the first part of this thesis, in which we directly address decision problems over Petri nets and VASS. We shall now turn our attention to one of the most common modern use cases for Petri nets: the verification of properties of concurrent systems.

In this chapter, we will introduce a new model of computation called *leafy automata*, or LAs for short. These LAs will provide a basis which we can use to interpret the semantics of a prototypical programming language Finitary Idealized Concurrent Algol (FICA). We will then constrain the LA model in a number of ways, which induces corresponding constraints on FICA. This will lead us to expressive fragments of FICA on which we can decide a family of model checking properties.

These motivations will be developed and expounded upon in later chapters. For now, we will introduce the model itself and discuss its structural and semantic properties.

6.1 Automata over infinite alphabets

Let us first define the notion of an automaton over an infinite alphabet. When dealing with "traditional" models of computation such as deterministic finite automata (DFAs), pushdown automata (PDAs) and Turing machines, the tokens that the machine will read are finite and known in advance. For example, a DFA over binary strings has the alphabet $\{0, 1\}$.

Automata defined over infinite alphabets do not have this guarantee. Instead, the letters will be pulled from some infinite set. In this thesis we shall be dealing with automata which operate over an infinite alphabet, but only have a finite number of transitions.

We may ask: "How can an automaton meaningfully work with an infinite alphabet, with only a finite number of transitions?" In short, transitions in automata over infinite alphabets will often operate via *predicates* over letters of the alphabet. An automaton that accepted any natural number $n \in \mathbb{N}$ as a token might activate different transitions depending on whether the token were odd or even; but it would not have entirely different semantics for each different value of n.

Similarly, the semantics of the automaton might predicate its transitions both on the letter being read and its relation to the current configuration of the machine. It may also operate based on some outside knowledge of the letter, such as its relationship to other letters in the infinite alphabet. Leafy automata, the model we will define in this chapter, will use both of these features.

Finally, it sometimes behooves us to make use of "tags" for control flow *in addition to* the power of the infinite alphabet. To that end, we might prefer to have two alphabets, one more reminiscent of a classic finite automaton, usually denoted by Σ, and one infinite alphabet of the type described above, which we will choose to denote by $\mathcal{D}$.

For the avoidance of ambiguity in what follows, we will write *letter* or *tag*, denoted by t, when speaking of elements of the finite alphabet Σ. Similarly for the infinite alphabet $\mathcal{D}$, we will say *data value*, and denote such data values by d (and subscripted variants thereof). It is common to use such alphabets as a Cartesian product; we will call the product $\Sigma \times \mathcal{D}$ a *data alphabet* and its elements *data letters* of the shape $(t, d) \in \Sigma \times \mathcal{D}$. Sequences of these data letters, i.e. of type $(\Sigma \times \mathcal{D})^*$, will be called *data words*, and sets of such sequences *data languages*.

6.2 Data Trees & Data Forests

The operation of leafy automata is dependent on the infinite dataset that they operate over, and the structure that we choose to impose on that dataset.

Let us start with a technical definition of our structure. We will say that $\mathcal{D}$ is equipped with a function $\mathit{pred} : \mathcal{D} \to \mathcal{D} \cup \{\perp\}$, such that the following two conditions hold:

- Infinite branching: $\mathit{pred}^{-1}(\{x\})$ is countably infinite for all $x \in \mathcal{D} \cup \{\perp\}$.
- Well-foundedness: for any $d \in \mathcal{D}$, there is some $i \in \mathbb{N}$ such that $\mathit{pred}^{i+1}(d) = \perp$.

The function pred is the *parent function* over $\mathcal{D}$; that is, if $\mathit{pred}(d) = d'$, we will say that d' is the *parent* of d. Similarly, $\mathit{pred}^{-1}(d)$ will denote the *children* of d. For convenience, we will also define a new function, $level : \mathcal{D} \to \mathbb{N}$, such that $level(d)$ gives us the value i as defined above in the well-foundedness condition. This represents the "depth" of the value in the set. We will call d a *level-i* data value. Data values at level-0 are called *roots*.

The definition of *pred* gives us countably infinite branching and countably infinite depth. Because there are countably many roots, the final structure imposed on $\mathcal{D}$ is that of a countably infinite directed forest.

For notational convenience, and without loss of generality, we will distinguish a particular tree inside our countably-infinite forest. Let us write d_r for the root of this tree. Its direct children (the level-1 data values) will be denoted $d_0, d_1, d_2, \ldots$. The children of d_0 will be $d_{00}, d_{01}, d_{02}, \ldots$, and so on. The subscripted string uniquely identifies the path from the root. In this way, we are able to succinctly describe some subset of the data values of $\mathcal{D}$, such that the relationships between them are visually apparent.

6.3 Leafy Automata

We are now in a position to formally define leafy automata.

A leafy automaton is a nondeterministic automaton over data words, whose internal configuration is a tree. The automaton evolves by adding and removing leaves to/from its tree configuration in correspondence with the data letters that it reads. The automaton is said to *terminate* (or *succeed*) with respect to some data word if there is some sequence of transitions by which the root node is created on reading the first data letter; the configuration evolves while reading the word; and finally the root is deleted on reading the final letter. The automaton is said to *diverge* (or *fail*) if it is unable to progress according to any transition.

Note that we are reusing the notion of *level* in the definition of LA. There is a direct correspondence between the levels of a leafy automaton and the levels of $\mathcal{D}$; this relationship will become more apparent when we discuss the operation of the model.

The formal definition of LA is as follows.

Definition 6.1. A level-k leafy automaton (k-LA) is a quadruple $\mathcal{A} = \langle \Sigma, k, Q, \delta \rangle$, where

- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is a finite alphabet, partitioned into questions and answers;
- $k \geq 0$ is the level parameter;
- $Q = \sum_{i=0}^{k} Q^{(i)}$ is a finite set of states, partitioned into sets $Q^{(i)}$ of level-i

states;

- $\delta = \delta_{\mathsf{Q}} + \delta_{\mathsf{A}}$ is a finite transition relation, partitioned into question- and answer-related transitions;
- $\delta_{\mathsf{Q}} = \sum_{i=0}^{k} \delta_{\mathsf{Q}}^{(i)}$ are the question transitions paritioned into layers, where $\delta_{\mathsf{Q}}^{(i)} \subseteq Q^{[0,i-1]} \times \Sigma_{\mathsf{Q}} \times Q^{[0,i]}$ for $0 \leq i \leq k$;
- $\delta_{\mathsf{A}} = \sum_{i=0}^{k} \delta_{\mathsf{A}}^{(i)}$ are the answer transitions partitoned into layers, where $\delta_{\mathsf{A}}^{(i)} \subseteq Q^{[0,i]} \times \Sigma_{\mathsf{A}} \times Q^{[0,i-1]}$ for $0 \leq i \leq k$.

Let us break down this definition in detail.

- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is a finite alphabet, partitioned into questions and answers;

  The question letters, in Σ_{Q}, are those which correspond to transitions which add a leaf to the machine's internal configuration when fired. Symmetrically the answer letters in Σ_{A} correspond to those transitions which remove a leaf. This correspondence is total; there are no other letters, and no other types of transition, and hence *every* transition will add or remove a leaf in the tree. This notion is fundamental to our understanding of LA, at least in this formulation.

- $k \geq 0$ is the level parameter;

  k identifies the maximum depth (distance from the root) at which leaves can be added or removed. A machine with level parameter k which reads a data value of level greater than k will not proceed, since definitionally it will have no transitions which correspond to such a data value.

- $Q = \sum_{i=0}^{k} Q^{(i)}$ is a finite set of states, partitioned into sets $Q^{(i)}$ of level-i states;

  For each level we define the set of states that may be associated to nodes at that level. It was decided that states should be isolated to levels rather than shared across the entire machine for two reasons. Firstly, nodes at different levels perform entirely different roles in the machine, in comparison to nodes at the same level which are not meaningfully distinguishable. Secondly, minimising the number of possible states for any given node significantly reduces the computational load when trying to compute state spaces for the sake of model checking.

- $\delta = \delta_{\mathsf{Q}} + \delta_{\mathsf{A}}$ is a finite transition function, partitioned into question- and answer-related transitions;
- $\delta_{\mathsf{Q}} = \sum_{i=0}^{k} \delta_{\mathsf{Q}}^{(i)}$ are the question transitions paritioned into layers, where $\delta_{\mathsf{Q}}^{(i)} \subseteq Q^{[0,i-1]} \times \Sigma_{\mathsf{Q}} \times Q^{[0,i]}$ for $0 \leq i \leq k$;

- $\delta_{\mathsf{A}} = \sum_{i=0}^{k} \delta_{\mathsf{A}}^{(i)}$ are the answer transitions partitoned into layers, where $\delta_{\mathsf{A}}^{(i)} \subseteq Q^{[0,i]} \times \Sigma_{\mathsf{A}} \times Q^{[0,i-1]}$ for $0 \leq i \leq k$.

  The notation $Q^{[i,j]}$ denotes the set of all sequences of states

  $$(q^{(i)}, q^{(i+1)}), \ldots, q^{(j)} \in Q^{(i)} \times Q^{(i+1)} \times \ldots \times Q^{(j)}.$$

  We distinguish a special case where there is no sequence of states (either the root has not yet been created, or has been deleted). This occurs for question and answer transitions where $i = 0$ (and hence $j = -1$). In this case, we write $Q^{[0,-1]} = \{\dagger\}$.

  When explicitly writing question transitions in full, we will write them as

  $$(q^{(0)}, q^{(1)}, \ldots, q^{(i-1)}) \xrightarrow{t_{\mathsf{Q}}} (r^{(0)}, r^{(1)}, \ldots, r^{(i-1)}, r^{(i)}).$$

  The same applies symmetrically for answer transitions. (Side note: observe the distinction between Q used to denote "question" and q/Q used for states.)

6.3.1 Configurations and Runs

As described before, the configurations of a leafy automaton are structured as a tree. Let us now formalise this notion.

A configuration of an LA is a triple $\kappa = (D, E, f)$. D is a finite subset of $\mathcal{D}$, that consists of those data values that have been encountered so far. The elements of D will always label some finite rooted subtree of $\mathcal{D}$. (In our examples, this subtree will be rooted at d_r.) E is a finite subset of D, corresponding to those elements of D that are still present in the current configuration. $f : E \to Q$ maps level-i data values onto level-i states. Formally, $\forall d \in E : \mathit{level}(d) = i \implies f(d) \in Q^{(i)}$. We will generalise f so that it also operates over sequences of data values: $f(d_0, d_1, \cdots, d_n) \stackrel{\text{def}}{=} (f(d_0), f(d_1), ...f(d_n))$.

The initial configuration is $\kappa_0 = (\emptyset, \emptyset, \emptyset)$. Final (terminal) configurations are of the shape $(D_t, \emptyset, \emptyset)$ for some D_t. Evolution of the configuration κ will proceed according to δ. As noted before, every transition will either add, or remove, a single leaf from the configuration. Progress can be made only in the following circumstances:

1. A data letter (t, d) is read such that $t \in \Sigma_{\mathsf{Q}}$ and $d \notin D$ and $\mathit{pred}(d) \in E$. That is, the letter is a question letter; the data value is fresh (has not been seen before); and its parent is present in the tree.

   Active transitions are those ones whose first component is equal to the sequence of states annotating (by f) the data values on the branch from the root to $\mathit{pred}(d)$ inclusive. Taking $i = \mathit{level}(d)$, for every transition $(s,t,s') \in \delta_{\mathsf{Q}}^{(i)}$ we have

$$(s,t,s') \text{ is active} \Longleftrightarrow s = f(\mathit{pred}^i(d), \cdots, \mathit{pred}(d)).$$

When activated, a transition rewrites the entire branch from the root to $\mathit{pred}(d)$ according to its third component, and adds d as a new leaf as a child of $\mathit{pred}(d)$. Formally, the transition moves the machine from (D,E,f) to $(D \cup \{d\}, E \cup \{d\}, f')$ where f' satisfies the following conditions:

- $(f(\mathit{pred}^i(d),\ldots,\mathit{pred}(d))\ ,\ t\ , f'(\mathit{pred}^i(d),\ldots,d)) \in \delta_{\mathsf{Q}}^{(i)}$;
- $\mathsf{dom}(f') = \mathsf{dom}(f) \cup \{d\}$; and
- $f'(x) = f(x)$ for all $x \notin \{\mathit{pred}(d),\ldots,\mathit{pred}^i(d)\}$.

2. On reading a letter (t,d) with $t \in \Sigma_{\mathsf{A}}$, $d \in E$, and $\mathit{pred}^{-1}(\{d\}) \cap E = \emptyset$. That is, we read an answer letter which is accompanied by a data value which is currently a leaf in the tree.

   Symmetrically with question letters, here the fireable transitions will be those whose first component matches the projection of f onto the branch from the root to d inclusive. So for every transition $(s,t,s') \in \delta_{\mathsf{A}}^{(i)}$ we have
$$(s,t,s') \text{ is active} \Longleftrightarrow s = f(\mathit{pred}^i(d), \cdots, d).$$

   A transition of this type will delete d from configuration, and at the same time update the branch from the root to $\mathit{pred}(d)$. Formally, we move from (D,E,f) to $(D, E - \{d\}, f')$, where f' satisfies:

   - $(f(\mathit{pred}^i(d), \cdots, d)\ ,\ t\ , f'(\mathit{pred}^i(d), \cdots, \mathit{pred}(d))) \in \delta_{\mathsf{A}}^{(i)}$;
   - $\mathsf{dom}(f') = \mathsf{dom}(f) \setminus \{d\}$; and
   - $f'(x) = f(x)$ for all $x \notin \{\mathit{pred}(d),\ldots,\mathit{pred}^i(d)\}$.

3. The first and last moves of the machine are treated specially, because $\mathit{pred}(d_r) = \bot$. On reading (t,d) as the first letter, we may progress from $(\emptyset,\emptyset,\emptyset)$ to $(\{d\},\{d\},\{d \mapsto r^{(0)}\})$ for every $\dagger \xrightarrow{t} r^{(0)} \in \delta_{\mathsf{Q}}^{(0)}$. The last move is similar; we move from $(D \uplus \{d\}, \{d\}, \{d \mapsto q^{(0)}\})$ to $(D,\emptyset,\emptyset)$ for every $q^{(0)} \xrightarrow{t} \dagger \in \delta_{\mathsf{A}}^{(0)}$. Observe that a given data word will always be rejected by an LA if its first and last data letters have different data values.

Observe that when the configuration of the machine changes, only the branch of the tree ending at d (the path from the root to d) will have its states changed; the rest of the configuration is unchanged. This gives rise to a notion of independence which will be explored later when discussing variants of LA.

Example 6.1. We will now go through a small example of leafy automaton and investigate how it reads a short data word. This will showcase the basic functionality of leafy automata and some of the notation that we shall use throughout the remainder of the thesis.

We will use the following notation for defining the transitions of the automaton. All other parameters and values should be assumed to be minimal with respect to the given set of transitions.[1]

$$\dagger \xrightarrow{\mathsf{start}} 0 \qquad \dagger \xrightarrow{\mathsf{start}} 1 \qquad 2 \xrightarrow{\mathsf{end}} \dagger$$

$$x \xrightarrow{\mathsf{add}} (x, \bullet) \qquad (1, \bullet) \xrightarrow{\mathsf{del}} 2$$

The maximum depth used for any state or transition in this machine is 1 (level-1 states appear in the add and del transitions), and hence this is a 1-LA. We can infer also that $Q^{(0)} = \{0, 1, 2\}$ and $Q^{(1)} = \{\bullet\}$.

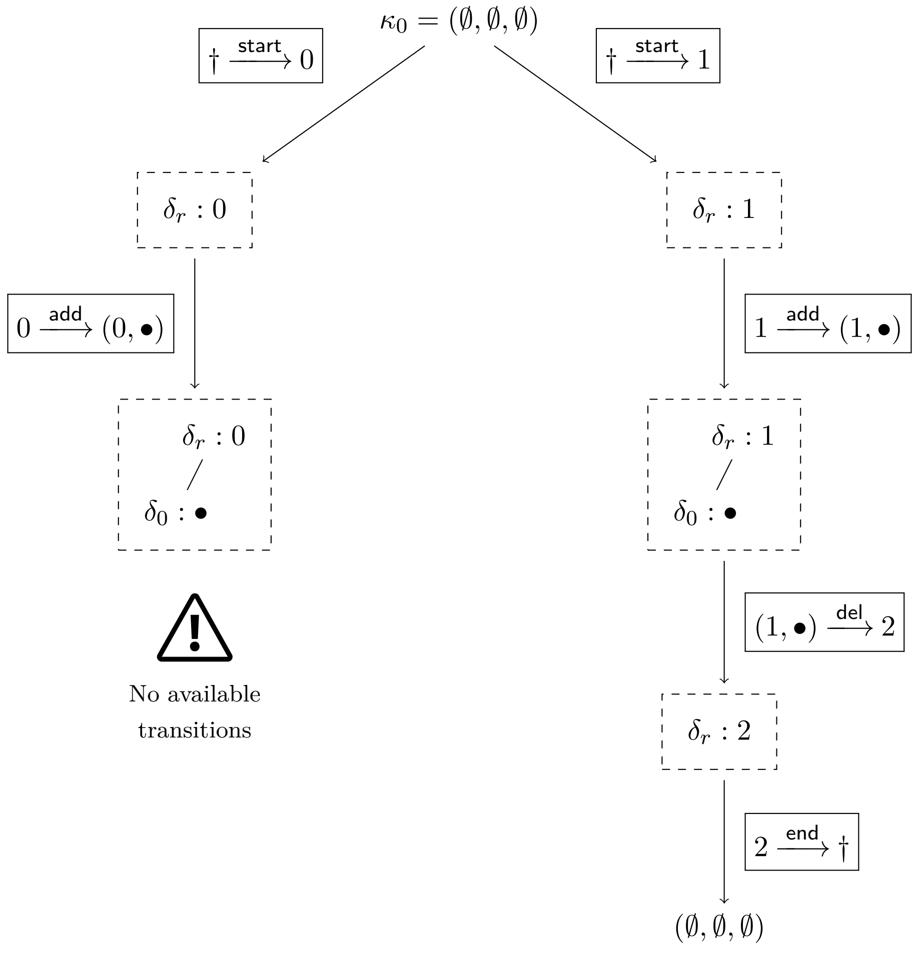
As alluded to previously, $\dagger$ is a distinguished state which represents the absence of any branch to update, and is functionally indistinguishable from the empty sequence. We also write 0 for (0), i.e. we intentionally confuse[2] a state with a length-1 sequence containing only that state. Thus x in the above example is shorthand for (x), a placeholder for (0), (1) or (2). Note that by the definition of LA, such single-element sequences always refer to the state at the root. Hence in this case, the first move may put the root into state 0 or 1; and the root may only be deleted (and the machine accept) if it is in state 2 when end is read. Hence start is a question transition (in δ_{Q}) and end is an answer transition (in δ_{A}).

For compactness of representation, it is convenient to denote some states in these sequences by variables. For example, the move labeled by add uses the variable x. The x is a wild-card and will match any state. Since x also appears in the right-hand side of the same transition, we may read this as saying that on reading tag add, the machine may add a new child to the root regardless of the current state of the root; and the state of the root will be preserved during that transition. So in this case, the notation would expand into two transitions: $0 \xrightarrow{\mathsf{add}} (0, \bullet)$ and $1 \xrightarrow{\mathsf{add}} (1, \bullet)$. Both transitions are questions, since they add a leaf when fired.

Lastly we have a delete transition tagged with del. The transition in ques-

[1]The bullet $\bullet$ is used as a singleton state where no useful information is conveyed.

[2]The phrase "intentionally confuse" in this context first appeared in the documentation for the Raku (a.k.a. Perl 6) programming language.



$\kappa_0 = (\emptyset, \emptyset, \emptyset)$
$\dagger \xrightarrow{\mathsf{start}} 0$
$\dagger \xrightarrow{\mathsf{start}} 1$
$\delta_r : 0$
$\delta_r : 1$
$0 \xrightarrow{\mathsf{add}} (0, \bullet)$
$1 \xrightarrow{\mathsf{add}} (1, \bullet)$
$\delta_r : 0$ / $\delta_0 : \bullet$
$\delta_r : 1$ / $\delta_0 : \bullet$
No available transitions
$(1, \bullet) \xrightarrow{\mathsf{del}} 2$
$\delta_r : 2$
$2 \xrightarrow{\mathsf{end}} \dagger$
$(\emptyset, \emptyset, \emptyset)$

Figure 6.1: The nondeterministic operation of a small leafy automaton over a short data word. The configurations are given in dashed boxes (as trees where possible) and labelled by the data values and states; transitions applied are in solid boxes.

tion will be able to delete any node at level 1 so long as it is in state 0 and its parent (the root) is in state 1. On activation the parent will be moved to state 2; any other children are unaffected. del is an answer move, as it removes a leaf when fired.

Let's now observe the nondeterministic operation of this machine over the following simple data word:

$$(\mathsf{start}, d_r)\ (\mathsf{add}, d_0)\ (\mathsf{del}, d_0)\ (\mathsf{end}, d_r)$$

In the diagram in Figure 6.1, we start in configuration $\kappa_0 = (\emptyset, \emptyset, \emptyset)$. The configuration tree is empty. Immediately on reading the data letter (start, d_r), the machine nondeterministically branches, applying $\dagger \xrightarrow{\mathsf{start}} 0$ in one branch (left) and $\dagger \xrightarrow{\mathsf{start}} 1$ in another branch (right).

On reading (add, d_0), the machine determines the level of d_0 (which is level

1) and tries to find transitions in $\delta_Q^{(1)}$ that meet the requirements. Both the left and right branches find different transitions that fit the preconditions present in their configurations and proceed accordingly. Since d_0 is the child of d_r, we know the child leaf will be added to the node labelled by d_r.

Now when reading (del, d_0), the left branch has no applicable transition, since the state path $(0, 0)$ from which d_0 is being removed does not match any transition's precondition. The nondeterministic branch dies. However the right thread's state path $(1, 0)$ does match a transition tagged with del, so it can progress, moving the root to state 2 in the process. Finally the end transition $2 \xrightarrow{\mathsf{end}} \dagger$ can be fired on reading (end, d_r) and the root node labelled by d_r is removed. With no more to read, the machine accepts the word.

As a careful reader might surmise by this point, leafy automata will often reject words out-of-hand based on their structure. If a word is not "well-formed" by some criteria, it cannot possibly be accepted in *any* LA. The structural properties of such well-formed words map neatly onto certain applications, which will be discussed in future sections. The notion of well-formedness is also relevant when we discuss traces, which we shall proceed to now.

6.4 Traces

Traces are data words which exist with respect to valid evolutions of the configuration of a particular leafy automaton. So, for an LA $\mathcal{A} = (\Sigma, k, Q, \delta)$, the traces of $\mathcal{A}$ will be those data words $w = (l_1, l_2, \cdots, l_{h-1}, l_h)$ such that

$$(\emptyset, \emptyset, \emptyset) \xrightarrow{l_1} \kappa_1 \xrightarrow{l_2} \cdots \xrightarrow{l_{h-1}} \kappa_{h-1} \xrightarrow{l_h} \kappa_h.$$

We say that a trace is *complete* if and only if $\kappa_h = (D, \emptyset, \emptyset)$ and $h > 0$; that is, a complete trace is both nonempty and ends with the deletion of the root node.

Example 6.2. Consider the 1-LA $\mathcal{A}$ with transitions defined thus:

$$\dagger \xrightarrow{\mathsf{start}} 0 \qquad 0 \xrightarrow{\mathsf{inc}} (0, \bullet) \qquad (0, \bullet) \xrightarrow{\mathsf{dec}} 0 \qquad 0 \xrightarrow{\mathsf{end}} \dagger$$

The names chosen to label transitions are arbitrary tags; here we use inc and dec as those reflect the actions occurring in the model we are simulating. In this machine the inc and dec moves do not change the state at the root. The effect is that those moves may be performed arbitrarily many times. In this case they may even be interleaved, so inc moves may proceed and follow dec moves and vice versa. In terms of the tree structure, inc will add a node at

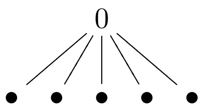


Figure 6.2: A configuration of a 1-LA simulating a one-counter. In the configuration pictured, the counter is storing the value 5.

level 1 and dec will remove such a node. end may only be fired once the root is a leaf, i.e. every inc move in the trace has been followed later in the trace by a corresponding dec move tagged with the same data value.

Taken together this means that the complete traces of $\mathcal{A}$ correspond to valid histories of a single nonnegative counter (those sequences of increments and decrements such that the value starts and ends at zero, and does not drop below zero). In this case, all traces are prefixes of some complete trace. This is not true for all LAs—the LA given in the previous example has traces which are not prefixes of accepted words. Figure 6.2 shows one configuration of the automaton. The branching degree of the root node in the configurations of this automaton corresponds to the value being "stored" in the counter; regardless of the number stored all of the nodes except the root are at level 1 and thus the tree is always of height at most 1.

6.4.1 Properties of traces

Traces will always obey certain structural properties.

Remark 6.1. *If a leafy automaton successfully reads a data letter (t_{Q}, d) for some $t_{\mathsf{Q}} \in \Sigma_{\mathsf{Q}}$ (and $level(d) > 0$), it must previously have encountered some data letter (t'_{q}, d') where $t'_{\mathsf{Q}} \in \Sigma_{\mathsf{Q}}$ and $d' = pred(d)$. Moreover it must* not *have yet encountered (t'_{A}, d') for any $t'_{\mathsf{A}} \in \Sigma_{\mathsf{A}}$.*

Remark 6.2. *When (t'_{A}, d') is successfully read, we know that every child $d \in pred^{-1}(d')$ which appeared previously in the trace has appeared twice: once with a question letter and secondly with an answer letter.*

Taken together, these two properties comprise a kind of "hierarchical bracketing"; that is, entities shallower in the tree must exist while their children exist. This hierarchical structure appears everywhere in computer science. We may imagine a similar structure of process trees: a process may only terminate once all of its child processes have terminated. For a very relevant example we may think about lexical scopes in a programming language: a lexical scope can be closed only after its use sites are done with it.

6.5 Complexity analysis of LA

With the model now fully introduced, we can investigate some of its theoretical properties, and why we believe leafy automata to be a good candidate to impose further restrictions on in the search for decidability.

We will start by taking a look at the *emptiness problem* for (k)-LA. The problem statement is simple: given an LA $\mathcal{A}$, is there no complete trace of $\mathcal{A}$? Or equivalently, does $\mathcal{A}$ reject every possible data word?

Theorem 6.1. *The emptiness problem for* 2-LA *is undecidable.*

Proof. We reduce from the halting problem on 2-counter machines to the emptiness problem for 2-LA.

Recall the construction from Example 6.2, whose traces correspond to valid histories on a nonnegative counter. We will extend this idea in order to represent an n-counter machine—for our purposes, a 2-counter machine—with an LA of depth 2.

For the sake of this proof, we shall consider a 2-counter machine to be a tuple (Q, q_0, q_F, δ) where Q is the set of states, q_0 and q_F single distinguished initial and final states, respectively, and $\delta \subseteq Q \times \{c_1, c_2\} \times \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero?}\} \times Q$ the transition function.

Consider a 2-LA whose two level-1 states are distinguished with states c_1 and c_2. The subtrees of these level-1 states will act as counters of the same shape as those in Example [something]. The tags on the level-1 states are such that each transition will only affect one counter or the other. The nodes at level 2 will have an arbitrary state. During operation, the root node's state will directly correspond to the state of the two-counter machine. In sum, the tree configuration of the 2-LA will be of the shape exemplified in Figure 6.3.

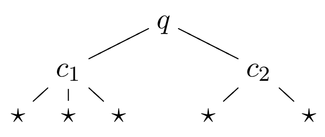


Figure 6.3: A configuration of our two-counter 2-LA. In the configuration pictured, counter c_1 has value 3 and counter c_2 has value 2.

The translation of transitions from a 2-counter machine into 2-LA is given in Figure 6.4. Increment and decrement of counter c (either c_1 or c_2) directly maps onto the addition and removal of children of the level-1 node in state c. In order to zero-test counter c we delete the node in state c and add a fresh one, also in state c. Of course this will only succeed if that node is a leaf in

$$\dagger \xrightarrow{\mathsf{start}} q_{\bot 1} \qquad q_1 \xrightarrow{\mathsf{start}_1} (q_{\bot 2}, c_1) \qquad q_{\bot 2} \xrightarrow{\mathsf{start}_2} (q_0, c_2)$$

$$(q_F, c_1) \xrightarrow{\mathsf{end}_1} q_{\top 1} \qquad (q_{\top 1}, c_2) \xrightarrow{\mathsf{end}_2} q_{\top 2} \qquad q_{\top 2} \xrightarrow{\mathsf{end}} \dagger$$

$$\frac{q \longrightarrow (c, \mathsf{inc}, q') \in \delta \qquad c \in \{c_1, c_2\}}{(q, c) \xrightarrow{\mathsf{inc_c}} (q', c, \star)} \qquad \frac{q \longrightarrow (c, \mathsf{dec}, q') \in \delta \qquad c \in \{c_1, c_2\}}{(q, c, \star) \xrightarrow{\mathsf{dec_c}} (q', c)}$$

$$\frac{q \longrightarrow (c, \mathsf{zero?}, q') \in \delta \qquad c \in \{c_1, c_2\}}{(q, c) \xrightarrow{\mathsf{zero?}} q_{\mathsf{zero,c,q'}} \qquad q_{\mathsf{zero,c,q'}} \xrightarrow{\mathsf{zero!}} (q', c)}$$

Figure 6.4: Translation rules which embed the behaviour of a 2-counter machine $\mathcal{M}$ with transition function δ into the moves of a 2-LA.

the la, i.e. it has no children and hence represents a value of 0; so a failed zero test will cause the automaton to reject as intended.

In order to support the translation into LA, we must provide a number of anciliary states at level 0. This is due to the restriction that only one branch of a leafy automaton may be modified at a time. In particular, note the sequence of intermediate introductory and final states $(q_{\bot 1}, q_{\bot 2})$ and $(q_{\top 1}, q_{\top 2})$. These are used as markers to enforce that valid traces must start with the sequence of letters $(\mathsf{start}, \mathsf{start}_1, \mathsf{start}_2)$ and end with the sequence $(\mathsf{end}_1, \mathsf{end}_2, \mathsf{end})$. This guarantees that the level-1 counters are set up and cleaned up correctly. Note also that the zero-testing transitions of the shape $q \xrightarrow{\epsilon} (c, \mathsf{zero?}, q')$ in $\mathcal{M}$ require a pair of related transitions in the LA to simulate them, as the counter node labelled by c must be deleted and recreated. This also requires an additional top-level state to ensure the next transition fired will recreate the counter and move to the appropriate state.

The reduction should now be clear. Given a two-counter machine $\mathcal{M}$ with alphabet Σ, we construct a 2-LA $\mathcal{A}_{\mathcal{M}}$ with the mechanism described above. $\mathcal{A}_{\mathcal{M}}$ will have a complete trace if and only if there is some word $w \in \Sigma^*$ such that $\mathcal{M}$ halts on reading w. $\square$

Recall the complexity class Ackermann from Section 3.1.2.

Theorem 6.2. *The emptiness problem for* 1-LA *is* Ackermann*-complete.*

Proof. The proof is in two parts. Firstly we give a exponential-time reduction from emptiness of 1-LA to reachability in Vector Addition Systems with States (VASS); and then we will given a polynomial-time reduction in the other direction. Per the results of [30, 102, 104], we get Ackermann-completeness of emptiness for 1-LA.

We start with the translation from a VASS $\mathcal{V}$ in dimension d to a 1-LA $\mathcal{A}_{\mathcal{V}}$. As in Theorem 6.1, we represent values in unary, but in this case we maintain all leaves at level 1. The state of each level-1 leaf will be drawn from $\{1, \ldots, d\}$. Adding a value of 1 to component i in the VASS is achieved by adding a new level-1 leaf in state i; subtraction corresponds to removing a leaf in the same way.

VASS updates are given in binary, and so this construction induces an exponential blow-up: an update of n bits in one component of the vector will require up to 2^n transitions in the LA. In order to accommodate this, additional transitory states are introduced at level 0. Suppose the VASS is of dimension 2 and includes a transition $t = (q, (-1, 3), r)$. Then the LA would need four transitions (with three transitory states) at level 0, which would manage the deletion of one level-1 child in state 1 and the addition of three further children in state 2.

Translating from a 1-LA to a VASS is fairly straightforward. The state at level-0 in the LA can be modelled as a triple of coordinates in the VASS, using the same encoding as in Theorem 3.1. For each level-1 state we assign one coordinate, such that the value of that coordinate in a configuration of the VASS corresponds to the number of level-1 children in that state in a configuration of the LA. Nonemptiness of the LA can then be modelled as reachability to a marking which is zero in all places except the triple of coordinates representing the machine's state, which will encode some distinguished end state. ☐

Theorem 6.3. *The equivalence problem for* 1-LA *is undecidable.*

Proof. We proceed by reducing from the halting problem for deterministic two-counter machines, which was shown to be undecidable in [113, pp. 255–258].

The input to this problem is a deterministic two-counter machine $\mathcal{C} = (Q_{\mathcal{C}}, q_0, q_F, T)$, where

- $Q_{\mathcal{C}}$ is the set of states;
- $q_0, q_F \in Q_{\mathcal{C}}$ are the initial and final state, respectively;
- $T : Q_{\mathcal{C}} \setminus \{q_F\} \to (\mathsf{INC} \cup \mathsf{JZDEC})$ is the step function.

Steps in INC are of the form $(i, q') \in \{1, 2\} \times Q_{\mathcal{C}}$ (increment counter i and go to state q'). Steps in JZDEC are of the form $(i, q', q'') \in \{1, 2\} \times Q_{\mathcal{C}} \times Q_{\mathcal{C}}$ (if counter i is 0 then go to state q'; else decrement counter i and go to state q''). The question is whether, starting from q_0 with both counters zero, $\mathcal{C}$ eventually reaches q_F with both counters zero.

We first construct a 1-LA that recognises the language of all data words such that the following two properties hold:

1. The underlying word (i.e., the projection onto the finite alphabet) encodes a path through the transition relation of $\mathcal{C}$ from the initial state to the final state. In other words, the word encodes a pseudo-run such that the non-negativity of counters and the correctness of zero tests are ignored.

2. The occurrences of the letters that encode increments and decrements of $\mathcal{C}$ form pairs that are labelled by the same level-1 data values, where each increment is earlier than the corresponding decrement. Assuming that both counters are zero initially, this ensures their non-negativity throughout the pseudo-run and their being zero at the end.

The second 1-LA is slightly more complex. It accepts data words that have the same properties as those accepted by the first 1-LA, and in addition:

3. There exists some increment followed by a zero test of the same counter before a decrement with the same data value has occurred. In other words, there is at least one incorrect zero test in the pseudo-run.

The two sets of accepted traces will be equal if and only if all pseudo-runs that satisfy the initial, non-negativity and final conditions necessarily contain some incorrect zero test, i.e. if and only if $\mathcal{C}$ does not halt as required. We give the formal construction below.

The two LAs we compute are $\mathcal{A}_{\mathcal{C}1} = \langle \Sigma, 1, Q, \delta_1 \rangle$ and $\mathcal{A}_{\mathcal{C}2} = \langle \Sigma, 1, Q, \delta_2 \rangle$.

The alphabet, $\Sigma = \Sigma_{\mathsf{Q}} \cup \Sigma_{\mathsf{A}}$, is defined as follows:

$$\Sigma_{\mathsf{Q}} = \{\mathsf{start}, \mathsf{inc}_1, \mathsf{inc}_2, \mathsf{zero}_1, \mathsf{zero}_2\} \qquad \Sigma_{\mathsf{A}} = \{\mathsf{end}, \mathsf{dec}_1, \mathsf{dec}_2, \mathsf{zero}'_1, \mathsf{zero}'_2\}$$

Traces of $\mathcal{A}_{\mathcal{C}1}$ and $\mathcal{A}_{\mathcal{C}2}$ represent pseudo-runs of $\mathcal{C}$, i.e. sequences of steps of the machine. Aside from start and end, each letter in the trace corresponds to the machine performing either an INC step (inc), the "then" of a JZDEC step (zero), or the "else" of a JZDEC step (dec). The zero' transition is a necessity which allows us to erase leaves added by zero. Each of inc, dec, zero, and zero' has two variants which encode i, the counter number in the corresponding step. We will say that two letters *match* if they are accompanied by the same data value.

By construction $\mathcal{A}_{\mathcal{C}1}$ will accept exactly the data words with the following properties, which correspond to the high-level description of our first 1-LA:

- The first letter in the trace is start and the last is a matching end.
- For each occurrence of $\mathsf{inc_i}$, there is a matching $\mathsf{dec_i}$ later in the trace.
- For each occurrence of $\mathsf{zero_i}$, there is a matching $\mathsf{zero'_i}$ later in the trace.
- The letters in the trace (excluding start and end) form a sequence $(a_0, \ldots, a_{n-1})$; there exists some sequence of states $(s_0, \ldots, s_n) \in Q_{\mathcal{C}}^{n+1}$ such that for all $i \in (0, \ldots, n-1)$, s_{i+1} appears as the second or third component of $T(s_i)$, and a_i is a step which may be performed at state s_i (irrespective of counter values).

The state space of the root, $Q^{(0)} = Q_{\mathcal{C}} \times \{\circ, \star, \mathbf{1}, \mathbf{2}\}$, comprises pairs where the first component corresponds to a state of $\mathcal{C}$ and the second tracks an observation of some invalid sequence. The second component is only used in $\mathcal{A}_{\mathcal{C}2}$. We denote the pair at the root by square brackets. The states of the leaves at level 1 are $Q^{(1)} = \bigcup \{ \ \{i, 0_i, i\star\} \mid i \in \{1, 2\} \ \}$, where 0_i denotes a temporary leaf generated by $\mathsf{zero_i}$, i denotes a counter, and $i\star$ denotes a counter being observed in $\mathcal{A}_{\mathcal{C}2}$.

The transition function δ_1 of $\mathcal{A}_{\mathcal{C}1}$ is defined as follows.

$$\dagger \xrightarrow{\mathsf{start}}{}_1 [q_0, \circ] \qquad [q_F, \circ] \xrightarrow{\mathsf{end}}{}_1 \dagger \qquad \frac{q \xrightarrow{\mathsf{INC}} (i, q') \ \in T}{[q, \circ] \xrightarrow{\mathsf{inc_i}}{}_1 ([q', \circ], i)}$$

$$\frac{q \xrightarrow{\mathsf{JZDEC}} (i, q', q'') \ \in T}{([q, \circ], i) \xrightarrow{\mathsf{dec_i}}{}_1 [q'', \circ] \qquad [q, \circ] \xrightarrow{\mathsf{zero_i}}{}_1 ([q', \circ], 0_i)} \qquad \frac{q \in Q_{\mathcal{C}}}{([q, \circ], 0_i) \xrightarrow{\mathsf{zero'_i}}{}_1 [q, \circ]}$$

By construction $\mathcal{A}_{\mathcal{C}2}$ accepts exactly those traces of $\mathcal{A}_{\mathcal{C}1}$ where at least one $\mathsf{zero_i}$ letter occurs in between an $\mathsf{inc_i}$ letter and the matching letter $\mathsf{dec_i}$. In other words, the "then" of a JZDEC step has been taken while the counter was nonzero. This is not a legal step, and so such a trace does not represent a computation of $\mathcal{C}$. This implements the high-level description of our second 1-LA.

In order to accept a word, $\mathcal{A}_{\mathcal{C}2}$ must change the second component of the root's state from $\star$ to $\circ$. It does this by nondeterministically choosing to observe some INC transition. From here, it proceeds as in $\mathcal{A}_{\mathcal{C}1}$ until either it meets the matching JZDEC, in which case the automaton rejects, or it meets an ifz transition on the same counter, at which point it marks the second component with $\circ$ and proceeds as in $\mathcal{A}_{\mathcal{C}1}$.

The transition function δ_2 of $\mathcal{A}_{\mathcal{C}2}$ is defined as follows:

$$\dagger \xrightarrow{\mathsf{start}}_2 [q_0,\star] \qquad [q_F,\circ] \xrightarrow{\mathsf{end}}_2 \dagger \qquad \frac{q \xrightarrow{\mathsf{INC}} (i,q') \in T \qquad x \in \{\circ,\star,\mathbf{1},\mathbf{2}\}}{[q,x] \xrightarrow{\mathsf{inc_i}}_2 ([q',x],i) \qquad [q,\star] \xrightarrow{\mathsf{inc_i}}_2 ([q,\mathbf{i}],i\star)}$$

$$\frac{q \xrightarrow{\mathsf{JZDEC}} (i,q',q'') \in T \qquad x \in \{\circ,\star,\mathbf{1},\mathbf{2}\}}{[q,x] \xrightarrow{\mathsf{zero_i}}_2 ([q',x],0_i) \qquad [q,\mathbf{i}] \xrightarrow{\mathsf{zero_i}}_2 ([q',\circ],0_i)} \qquad \frac{q \in Q_{\mathcal{C}} \qquad x \in \{\circ,\star,\mathbf{1},\mathbf{2}\}}{([q,x],0_i) \xrightarrow{\mathsf{zero'_i}}_2 [q,x]}$$

$$\frac{q \xrightarrow{\mathsf{JZDEC}} (i,q',q'') \in T \qquad x \in \{\circ,\star,\mathbf{1},\mathbf{2}\}}{([q,x],i) \xrightarrow{\mathsf{dec_i}}_2 [q'',x] \qquad ([q,\circ],i\star) \xrightarrow{\mathsf{dec_i}}_2 [q'',\circ]}$$

$\mathcal{A}_{\mathcal{C}1}$ captures every correctness condition for halting computations of $\mathcal{C}$ except the legality of zero steps. Hence, $\mathcal{A}_{\mathcal{C}2}$ accepts exactly those accepted traces of $\mathcal{A}_{\mathcal{C}1}$ which are *not* halting computations of $\mathcal{C}$, and so $\mathcal{C}$ performs a halting computation if and only if $\mathcal{A}_{\mathcal{C}1} \neq \mathcal{A}_{\mathcal{C}2}$.

$\square$

The above results are a smörgåsbord of undecidability and computational intractability. From the perspective of model checking, this is not encouraging. However, the leafy automaton lends itself nicely to a variety of restrictions which make for a computational model both strongly representative of its domain and amenable to verification. It is these that we shall investigate in later chapters. First though, we shall discuss the language `FICA` and its relationship to leafy automata.

Chapter 7

FICA

In this chapter, we shall discuss the prototypical language Finitary Idealized Concurrent ALGOL (FICA). In particular, we shall identify what properties of the language make it a valuable substrate for programming language theory. Later, we investigate how our Leafy Automaton construct, set out in Chapter 6, can be employed to make FICA more conducive to model checking and computer-aided verification.

For a history of FICA, refer to Section 2.5. Here we shall dive straight into the definitions and applications of the language.

7.1 The FICA Language

As outlined above, Finitary Idealized Concurrent Algol (FICA) is a language with a variety of syntactic constructs for different methods of programming, including higher-order and imperative programming constructs, and control flow for sequential and parallel computation.

The types of FICA are defined by the following simple grammar:

$$\theta ::= \beta \mid \theta \to \theta \qquad \beta ::= \mathbf{com} \mid \mathbf{exp} \mid \mathbf{var} \mid \mathbf{sem}$$

com is the type of commands (that is, computations which do not return a value); **exp** is the type of expressions, whose values are in the range $\{1, \ldots, max\}$; **var** is the type of variables (the "names" in our call-by-name semantics); and **sem** is the type of binary semaphores.

The full typing rules for FICA are given in Figure 7.1. The imperative constructs should be familiar: **if-then-else** and **while** are used to implement branching and iteration, respectively; observe that branches of **if-then-else** must have the same type and the main block of the **while** loop must be of type **com**. Variables are declared with **newvar-in**, and assigned to with the (:=)

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{com}} \qquad \frac{}{\Gamma \vdash \mathbf{div}_\theta : \theta} \qquad \frac{}{\Gamma \vdash i : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{exp}}{\Gamma \vdash \mathbf{op}(M) : \mathbf{exp}}$$

$$\frac{\Gamma \vdash M : \mathbf{com} \qquad \Gamma \vdash N : \beta}{\Gamma \vdash M; N : \beta} \qquad \frac{\Gamma \vdash M : \mathbf{com} \qquad \Gamma \vdash N : \mathbf{com}}{\Gamma \vdash M || N : \mathbf{com}}$$

$$\frac{\Gamma \vdash M : \mathbf{exp} \qquad \Gamma \vdash N_1, N_2 : \beta}{\Gamma \vdash \mathbf{if}\, M\, \mathbf{then}\, N_1\, \mathbf{else}\, N_2 : \beta} \qquad \frac{\Gamma \vdash M : \mathbf{exp} \qquad \Gamma \vdash N : \mathbf{com}}{\Gamma \vdash \mathbf{while}\, M\, \mathbf{do}\, N : \mathbf{com}}$$

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x.M : \theta \rightarrow \theta'} \qquad \frac{\Gamma \vdash M : \theta \rightarrow \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

$$\frac{\Gamma \vdash M : \mathbf{var} \qquad \Gamma \vdash N : \mathbf{exp}}{\Gamma \vdash M := N : \mathbf{com}} \qquad \frac{\Gamma \vdash M : \mathbf{var}}{\Gamma \vdash !M : \mathbf{exp}}$$

$$\frac{\Gamma \vdash M : \mathbf{sem}}{\Gamma \vdash \mathbf{release}(M) : \mathbf{com}} \qquad \frac{\Gamma \vdash M : \mathbf{sem}}{\Gamma \vdash \mathbf{grab}(M) : \mathbf{com}}$$

$$\frac{\Gamma, x : \mathbf{var} \vdash M : \mathbf{com}, \mathbf{exp}}{\Gamma \vdash \mathbf{newvar}\, x := i\, \mathbf{in}\, M : \mathbf{com}, \mathbf{exp}} \qquad \frac{\Gamma, s : \mathbf{sem} \vdash M : \mathbf{com}, \mathbf{exp}}{\Gamma \vdash \mathbf{newsem}\, s\, \mathbf{in}\, M : \mathbf{com}, \mathbf{exp}}$$

Figure 7.1: `FICA` typing rules

operator. To dereference a variable, i.e. to get the value currently assigned to a name, it is prefixed by (!). Statements are sequenced with (;), as in many modern programming languages.

Function abstraction and function application are reminiscent of the typed lambda calculus. We assume that the only mathematical operators **op** are **succ** for successor and **pred** for predecessor; as we take the type of expressions in `FICA` to be finite, we can construct more exciting mathematical operators with conditionals.

Parallel composition of terms is achieved with the (||) operator. Semaphores are introduced with **newsem**, with the same syntax as **newvar** but without an initial value. Semaphores are manipulated with two new primitives: **grab**(s) will block a thread until semaphore s is free and take control of it; **release**(s) will release semaphore s under that thread's control. Two further concurrency primitives are also introduced: **skip** represents a trivially terminating computation, and **div** a trivially diverging computation.

7.1.1 Contextual Equivalence

One point of note is that well-formed `FICA` terms may contain free variables: variables not explicitly introduced via **newvar** within the term. Such a term is called *open*. The existence of open terms gives rise to a notion of *contextual dependence*: an open term's semantics may change depending on the context that the term is evaluated with. This notion of contexts is strongly related to the game-semantic view of `FICA`. We are most interested in *contextual* *equivalence*: whether two terms are indistinguishable in every context.

Ghica, Murawski and Ong showed in [71] that the problem of contextual equivalence for `FICA` is already undecidable with only second-order functions and without recursion. This gives us some indication of the amount of restriction that we must impose to achieve decidability of contextual equivalence in general. In this work, we choose to focus on a specific instantiation of contextual equivalence, which is equivalence with **div**. If a term is contextually equivalent to **div**, then there is no context in which the term is used and in which it terminates. There is a natural parallel here with automata-theoretic emptiness problems, and indeed we will see that correspondence borne out shortly.

Contextual equivalence with **div** still gives us the power to perform verification of a large class of properties. In particular, for some open term M, we can determine whether there is a context for M which leads to an undesirable state.

Example 7.1. Suppose that we have some term $x : \mathbf{var} \vdash M : \theta$ which makes use of a free variable x. (We use the single turnstile $\vdash$ to indicate a typing context; the term here is called M, and it has type θ.) The following term will be contextually equivalent to **div** if and only if x is never set to 13 during a terminating execution:

$$f : \theta \to \mathbf{com} \vdash \mathbf{newvar}\, x := 0\, \mathbf{in}\, (f(M) \,||\, \mathbf{if}\, !x = 13\, \mathbf{then\ skip\ else\ div})$$

Observe that f is free, and may be arbitrarily complex in how it uses its argument M (including using it across multiple threads, and unboundedly many times). f is left to be determined by the context; this should give some indication of the power of quantification over all contexts.

7.2 Game Semantics

In order to unify our automata-theoretic model of computation with the programming language theory of `FICA`, we shall introduce a third foundational theory: *game semantics*. Game semantics is a field which uses the theoretic presentations of games to model the logic of other branches of mathematics. We present here only a distilled summary of game semantics as it relates to `FICA`, the majority of which was first outlined in [70].

In game semantics, it is typical that games are dialogues between exactly two players. The classical notions of winners or payoffs are not of interest; the “players” are not viewed as in direct competition, but as two actors moving the state of play forward according to their own available move sets.

The Proponent player, written **P**, can be viewed as the "evaluator" of a term. The moves that they take will correspond to the shape of the term. The Opponent (**O**) represents the context that the term is evaluated in. While in the sequential setting it is expected that players alternate, when modelling concurrency this is no longer true—valid sequences of moves may see either player taking many moves in a row.

7.2.1 Arenas

These dialogue games take place in an *arena*, defined below.

Definition 7.1 (Arenas)**.** An *arena* is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$, where:

- M_A is a set of *moves*;
- $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a function which, for each move $m \in M_A$, determines whether it is an Opponent or Proponent move, and a question or an answer move; we write λ_A^{OP} and λ_A^{QA} for the first and second projections over λ_A; and
- $\vdash_A$ is the "enabling relation" over M_A, which satisfies the following:
  1. if there is no m such that $m \vdash_A n$, then $\lambda_A(n) = (O, Q)$;
  2. if $m \vdash_A n$ then $\lambda_A(m) \neq \lambda_A(n)$; and
  3. if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$.

If $m \vdash_A n$, then we say m *enables* n. The moves which are enabled by no move are called *initial* moves, and the set of such moves is denoted I_A. (Observe that all such moves are Opponent question moves, per condition (1) on the enabling relation in Definition 7.1.) The arena corresponding to some type θ is written $[\![\theta]\!]$. We will write *O-move* and *P-move* for moves belonging to the Opponent or the Proponent, respectively. If those moves are known to be question or answer moves, we may instead call them *O-questions* and *O-answers* (and likewise for P).

For any given term, there is some arena which interprets that term. The shape of moves permitted for each player is dictated by the type(s) present in the term.

Arenas are defined inductively. To interpret a term of a given type θ, we construct the arena for the sub-types of θ and unify them using one of two constructions.

The base case of this induction is at the base types of the language: **com**, **var**, **exp**, and **sem**. Each of these has an arena corresponding to it. In these

Arena	O-question	P-answers	Arena	O-question	P-answers
$[\![\mathbf{com}]\!]$	run	done	$[\![\mathbf{exp}]\!]$	q	i
$[\![\mathbf{var}]\!]$	read	i	$[\![\mathbf{sem}]\!]$	grb	ok
	write(i)	ok		rls	ok

Table 7.1: The moves in arenas corresponding to the base types of `FICA`. The value of i may be any value of type **exp** ($i \in \{0, \cdots, max\}$.)

arenas, all questions are initial O-questions, and all answers are P-answers. The moves of the arena are given in Table 7.1.

Arenas for types defined on these base types, through the judgements given in Figure 7.1, are interpreted using the *product* ($A \times B$) and *arrow* ($A \Rightarrow B$) constructions, given in Figure 7.2.

$$\begin{array}{rcl} M_{A\times B} & = & M_A + M_B \\ \lambda_{A\times B} & = & [\lambda_A, \lambda_B] \\ \vdash_{A\times B} & = & \vdash_A + \vdash_B \\ \\ M_{A\Rightarrow B} & = & M_A + M_B \\ \lambda_{A\Rightarrow B} & = & [\langle \lambda_A^{PO}, \lambda_A^{QA} \rangle, \lambda_B] \\ \vdash_{A\Rightarrow B} & = & \vdash_A + \vdash_B + \{\, (b,a) \mid b \in I_B \text{ and } a \in I_A \} \end{array}$$

Figure 7.2: Inductive rules for computing arenas for non-base `FICA` types. Note that $\lambda_A^{PO}(m) = O$ if and only if $\lambda_A^{OP}(m) = P$.

We will write $[\![\theta]\!]$ for the arena corresponding to some type θ.

Observe that unions in Figure 7.2 are expected to be disjoint, but multiple instances of the same move could appear in an inductively defined type. In order to distinguish between multiple copies of the same move, we shall modify the instances with a scheme of superscripts. All moves start off with an empty superscript. When an arena A is used as part of an inductive type, the prefixes of all of the moves in its move set M_A are prepended a new (1-indexed) natural number which uniquely identifies the arena at that level. For notational convenience we will omit separators between such integers, since in our examples every integer will be one digit.

The arena for a function type $\theta_A \to \theta_B$ uses the arrow construction:

$$[\![\theta_A \to \theta_B]\!] = [\![\theta_A]\!] \Rightarrow [\![\theta_B]\!].$$

Example 7.2. Consider the type $T_1 = \mathbf{com} \to \mathbf{com} \to \mathbf{com}$. We shall diagram the enabling relation $\vdash_{A_1}$ for the arena $A_1 = [\![\mathbf{com} \to \mathbf{com} \to \mathbf{com}]\!]$ corresponding to T_1.

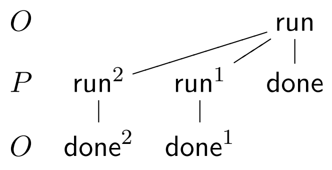


Figure 7.3: The enabling relation for $A_1 = [\![\mathbf{com} \to \mathbf{com} \to \mathbf{com}]\!]$.

Observe that the two instances of *run* are at the same level are distinguished by their different superscripted integers.

Example 7.3. Consider the type $T_2 = (\mathbf{var} \to \mathbf{com}) \to \mathbf{com}$. We shall diagram the enabling relation $\vdash_{A_2}$ for the arena $A_2 = [\![(\mathbf{var} \to \mathbf{com}) \to \mathbf{com}]\!]$ corresponding to T_2.

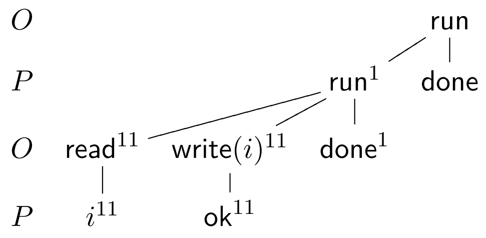


Figure 7.4: The enabling relation for $A_2 = [\![\mathbf{com} \to \mathbf{com} \to \mathbf{com}]\!]$.

In this case, the usage of **var** may involve reading or writing, and hence both O-moves read^{11} and $\mathsf{write}(i)^{11}$ are enabled by the Proponent playing run^1. The moves read^{11}, $\mathsf{write}(i)^{11}$, i^{11}, and ok^{11} all occur under two applications of arrow construction, and hence their superscripts are of length 2.

Indeed, by inductive construction, all question moves with (non-empty) superscript $s_1 \cdots s_{n-1}s_n$ will be enabled by some question move with (possibly-empty) superscript $s_1 \cdots s_{n-1}$; and answer moves with superscript s will be enabled by question moves with the same superscript s. This notion is visualised in Examples 7.2 and 7.3.

7.2.2 Plays

In order to derive meaning from arenas, we interest ourselves with sequences of moves made within them. We want to define a notion of a (legal) play: a sequence of moves which follows a structure corresponding to some valid interpretation of a term.

Consider a sequence of moves $m_0\, m_1\, \cdots\, m_n$. If that sequence of moves is to be considered a legal play, we want evidence that each move was valid at the time it was made. This evidence is given in the form of a *justification pointer*: a reference to some earlier move which enabled it. Formally:

Definition 7.2 (Justification)**.**

A *justified sequence* is a (non-empty) sequence $s = m_0\,(m_1, p_1)\,(m_2, p_2)\,\cdots$ in which every move after the first is justified.

A *justified* move is a pair (m_k, p_k) where m_k is a move and p_k identifies one move m_j in s such that $j < k$ and m_j enables m_k.

The precise mechanism by which we identify preceding moves by pointers will be described in Section 7.3. The first move is necessarily initial (from I_A), and therefore has no enabling move, so we leave it unjustified. When a move n is justified by a previous move m, we say that *m justifies n*. In the case where n is an answer, we say *n answers m*. A question move is called *pending* if it has not yet been answered—hence every question in a justified sequence is either pending or answered.

Not every justified sequence is a legal play: there are still requirements that such a sequence must fulfil in order to be considered legal. These requirements comprise a well-formedness condition: any started sub-computations must terminate before the parent computation terminates. This is formalised with the following pair of conditions.

Definition 7.3 (Legal plays)**.** The set P_A of *(legal) plays* over some arena A consists of the set of all finite justified sequences $m_0\,(m_1, p_1)\,(m_2, p_2)\,\cdots \in \lambda_A \cdot (\lambda_A \times \mathbb{N})^*$ satisfying both of the following conditions:

FORK: In any prefix $\cdots (m_i, p_i) \cdots (m_j, p_j)$ of s such that m_i justifies m_j, the (question) move m_i must be pending in $\cdots (m_i, p_i) \cdots (m_{j-1}, p_{j-1})$.

WAIT: In any prefix $\cdots (m_i, p_i) \cdots (m_j, p_j)$ of s such that m_j answers m_i, all questions justified by m_i must be answered.

Note that as only question moves enable other moves, all pointers must point to questions. Each question admits exactly one answer justified by it in any legal play, though it may also justify further question moves.

To obviate a full exposition of the pointer scheme here, and to aid in digestion of examples, when diagramming a justified sequence we will prefer to draw arrows between moves, such that an arrow pointing from move m_j back to move m_i indicates that the pointer p_j identifies m_i as the move which justified m_j.

Example 7.4. Pictured are two legal plays, the former over A_1 defined in Example 7.2 and the latter over A_2 defined in Example 7.3.

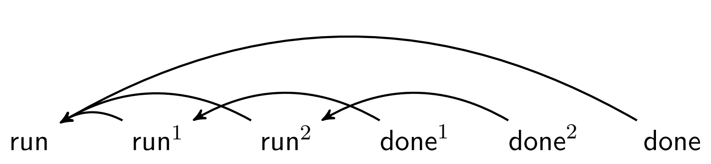


Figure 7.5: A justified sequence over arena A_1.

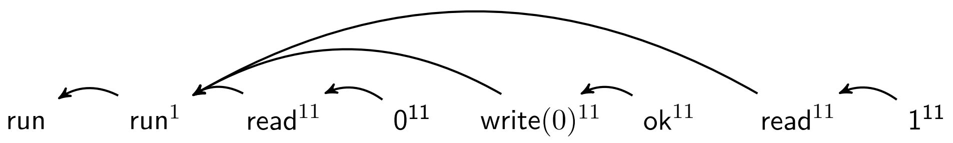


Figure 7.6: A justified sequence over arena A_2.

Both justified sequences in example 7.4 are legal plays, satisfying both the FORK and WAIT conditions. Observe that in the play in Figure 7.5, every question is answered. Such plays we call *complete*; there is no move which can legally extend the play. The justified sequence in Figure 7.6 is *not* complete, as there are pending questions. In particular, run and run_1 are both pending.

Observe also that the same move may appear multiple times in the same play. In this case, read^{11} is played more than once. This is a valid interpretation: with a **var** in argument position, a function provided by the context may read from and/or write to it arbitrarily many times.

Definition 7.4 (Strategies)**.**

A subset σ of the set P_A of plays over A is called *O-complete* if and only if $s \in \sigma$ and $s \cdot o \in P_A$ implies $s \cdot o \in \sigma$, for every O-move o.

A *strategy* σ over an arena A is a prefix-closed, O-complete subset of P_A. If σ is a strategy over A, we write $\sigma : A$.

The above is sufficient for defining the arenas for types, but we wish to evaluate terms *in context*. This requires us to define an arena corresponding to both a term, and the context in which the term is situated.

Let $\Gamma = \{x_1 : \theta_1, \ldots, x_l : \theta_l\}$ be a context, and $\Gamma \vdash M : \theta$ some FICA term in that context. We shall construct the arena for a term-in-context by constructing the whole as a function from context to the term's type. Hence $[\![\Gamma \vdash \theta]\!]$ is defined as $[\![\theta_1]\!] \times \ldots \times [\![\theta_l]\!] \Rightarrow [\![\theta]\!]$.

Ghica and Murawski show in [70] that a strategy can be assigned to the type of any FICA term-in-context $\Gamma \vdash M : \theta$. We will denote that strategy by

$[\![\Gamma \vdash M]\!]$. For example, we have that $[\![\Gamma \vdash \mathbf{div}]\!] = \{\epsilon, \mathsf{run}\}$, and $[\![\Gamma \vdash \mathbf{skip}]\!] = \{\epsilon, \mathsf{run}, \mathsf{run}\overset{\curvearrowleft}{\quad}\mathsf{done}\}$. We denote by $\mathsf{comp}(\sigma)$ the set of non-empty complete plays in σ. (Note that in particular $\mathsf{comp}([\![\Gamma \vdash \mathbf{div}]\!])$ is empty, since **div** cannot terminate.)

Two terms M_1 and M_2 are said to be *may-equivalent* ($\Gamma \vdash M_1 \cong_{may} M_2$) if and only if M_1 placed into any context Γ may terminate if and only if M_2 may terminate in the same context.

Definition (Full abstraction [112]). A model is *fully abstract* if and only if may-equivalence coincides with equality with respect to interpretations.

The definitions in this section are sufficient to give us full abstraction with respect to may-equivalence:

Theorem 7.1 (Full abstraction, [70]). *Per the game model given in this section,*

$$\Gamma \vdash M_1 \cong_{may} M_2 \text{ if and only if } \mathit{comp}([\![\Gamma \vdash M_1]\!]) = \mathit{comp}([\![\Gamma \vdash M_2]\!]).$$

Hence this game model is fully abstract.

As we will focus on equivalence with **div**, we will tend to be asking the decision problem of whether $\mathsf{comp}([\![\Gamma \vdash M_1]\!]) = \emptyset$.

7.3 From FICA to LA

With the game model cemented, it is time to investigate the relationship between terms of FICA (or more precisely, plays over those terms) and leafy automata. We shall start by showing that every FICA term has an LA whose complete traces correspond to complete plays of the term.

Let $T = \Gamma \vdash M : \theta$ be a term-in-context, and $\mathcal{A}$ be the automaton to construct. The first step is to define the finite alphabet that the $\mathcal{A}$ will read letters of.

Recall that the base moves of our game model are

$$\begin{aligned} \mathcal{M} &= M_{[\![\mathbf{com}]\!]} \cup M_{[\![\mathbf{exp}]\!]} \cup M_{[\![\mathbf{var}]\!]} \cup M_{[\![\mathbf{sem}]\!]} \\ &= \{\mathsf{run}, \mathsf{done}, \mathsf{q}, \mathsf{read}, \mathsf{grb}, \mathsf{rls}, \mathsf{ok}\} \cup \{i, \mathsf{write}(i) | 0 \leq i \leq max\}. \end{aligned}$$

The finite alphabet of the machine will correspond to these moves. As with the game model we shall be making use of a specially-crafted scheme of subscripts to distinguish between different instantiations of the same letter.

The scheme is similar, making use of a sequence of natural numbers to identify the provenance of a move; but we also encode justification pointers into the superscript, such that the enabling move can be recovered.

Suppose that $\Gamma = \{x_1 : \theta_1, \cdots, x_l : \theta_l\}$; that $\vec{i} \in \mathbb{N}^*$ is an integer sequence (called the *indexing vector*); and that $\rho \in \mathbb{N}$. Then the superscripts we use will be of one of two shapes, depending on the source of the move:

- Moves from the term's type θ will have superscripts $(\vec{i}, \rho)$; and
- Moves from each context type θ_j will have superscripts $(x_i, \vec{i}, \rho)$.

So for any move $m \in \mathcal{M}$, we shall write either $m^{(\vec{i},\rho)}$ or $m^{(x_i,\vec{i},\rho)}$. In the cases where $\rho = 0$, we may choose to omit it; likewise when $\vec{i} = \epsilon$. So a move m on its own is shorthand for $m^{(\epsilon,0)}$.

The following definition explains how the $\vec{i}$ superscripts are linked to moves from θ. For each move m, given that $X \subseteq \{m^{(\vec{i},\rho)} \mid \vec{i} \in \mathbb{N}^*, \rho \in \mathbb{N}\}$ and $y \in \mathbb{N} \cup \{x_1, \ldots, x_l\}$, let us write $y \cdot X$ for $\{m^{(y \cdot \vec{i})} \mid m^{(\vec{i},\rho)} \in X\}$. In words, $y \cdot X$ represents the prefixing of the indexing vectors of all superscripted moves of X with some index y. This procedure for inductively nesting moves was also described in Section 7.2.

Definition 7.5 (Term(-in-context) alphabets)**.** For any type θ, the corresponding alphabet $\mathcal{T}_\theta$ is defined as follows.

For base types $\beta = \mathbf{com}, \mathbf{exp}, \mathbf{var}$ or $\mathbf{sem}$:

$$\mathcal{T}_\beta = \{\, m^{(\epsilon,\rho)} \mid m \in M_{[\![\beta]\!]}, \rho \in \mathbb{N} \,\}.$$

For inductively defined types $\theta = \theta_h \to \cdots \to \theta_1 \to \beta$:

$$\mathcal{T}_\theta = \bigcup_{u=1}^{h} u \cdot T_{\theta_u} \cup \theta_\beta.$$

For a context $\Gamma = \{x_1 : \theta_1, \cdots, x_l : \theta_l\}$, the alphabet $\mathcal{T}_{\Gamma \vdash \theta}$ is defined as:

$$\mathcal{T}_{\Gamma \vdash \theta} = \bigcup_{v=1}^{l} x_v \cdot \mathcal{T}_{\theta_v} \cup \mathcal{T}_\theta.$$

Example 7.5. Consider a context $f : \mathbf{com} \to \mathbf{com}, x : \mathbf{com}$ for some term of type $\mathbf{com}$. The alphabet $\mathcal{T}_{f:\mathbf{com}\to\mathbf{com},x:\mathbf{com}\,\vdash\,\mathbf{com}}$ is

$$\left\{ \begin{array}{llll} \mathsf{run}^{(f1,\rho)}, & \mathsf{done}^{(f1,\rho)}, & \mathsf{run}^{(f,\rho)}, & \mathsf{done}^{(f,\rho)}, \\ \mathsf{run}^{(x,\rho)}, & \mathsf{done}^{(x,\rho)}, & \mathsf{run}^{(\epsilon,\rho)}, & \mathsf{done}^{(\epsilon,\rho)} \end{array} \middle|\, \rho \in \mathbb{N} \right\}.$$

The partitioning of these alphabets into question and answer letters exactly follows the definitions of question and answer moves from the game semantics.

The alphabets defined above are more than what we need to represent the game semantics of terms in context with LA. Indeed, we shall only use finite subsets of $\mathcal{T}_{\Gamma\vdash\theta}$ to represent a term $\Gamma \vdash \theta$, because the paramater ρ will be structurally bounded by the term itself, as we will now see.

Recall that configurations of LA have a tree structure, backed by the tree structure imposed on our infinite set of data values $\mathcal{D}$. The justification pointers used in the game-semantic world are encoded as a combination of the indexing vector and the value of ρ for that letter, and the data values that are read alongside that letter.

Occurrences of questions are represented directly with data values. An answer for a pending question is uniquely identified by the data value it is attached to: no auxiliary information is needed, and the pertinent question is trivially identified. (We leave ρ set to 0.) An answer not enabled by any question is illegal, as is an answer for an answered question; we do not need to represent those.

It remains to define the justification of question moves. Initial question moves have no pointer—these are also trivially identified, and we leave ρ set to 0. For non-initial questions, we use a combination of ρ and the attached data value. Note that, per Remark 6.1, when a leafy automaton reads some (non-level-0) data letter (t_i, d_i) as the next data letter of trace w, it must have previously read exactly one data letter (t', d') such that $\mathit{pred}(d) = d'$, and moreover t is a question move. We can extend this logic all the way back to the root: there must be some unique subsequence $(t_0, t_0) \cdots (t_{i-1}, d_{i-1})$ of w such that $\mathit{pred}(d_x) = d_{x-1}$, and all such d_x have appeared exactly once in the trace so far. We use ρ to mark which of these data letters justified the move. $\rho = 0$ indicates the parent (the move that was tagged with d_{i-1}) is the justifier; $\rho = 1$ the grandparent; and so on. In general, the justification pointer for non-initial question data letter $(t^{(\vec{i},\rho)}, i)$ goes to the data letter $(t_{i-\rho-1}, d_{i-\rho-1})$.

For certain purposes it shall become useful to have questions which do not correspond to a specific move. We will write ϵ_{Q} and ϵ_{A} for such question letters and answer letters; and transition rules making use of these will be such that ϵ_{A} must always succeed ϵ_{Q}. The introduction of these letters means that there may be multiple data words which represent the same play.

Example 7.6. Fix the following subtree of $\mathcal{D}$, rooted at d_0:

$$\begin{tikzcd}
& d_0 \arrow[d, dash] & \\
& d_1 \arrow[dl, dash] \arrow[dr, dash] & \\
d_2 \arrow[d, dash] & & d_2' \arrow[d, dash] \\
d_3 & & d_3'
\end{tikzcd}$$

and the following data word, w:

$$(\mathsf{run}, d_0)\,(\mathsf{run}^f, d_1)\,(\mathsf{run}^{f1}, d_2)\,(\mathsf{run}^{f1}, d_2')\,(\mathsf{run}^{(x,2)}, d_3)\,(\mathsf{run}^{(x,2)}, d_3')\,(\mathsf{done}^x, d_3)$$

Then w represents the following play.

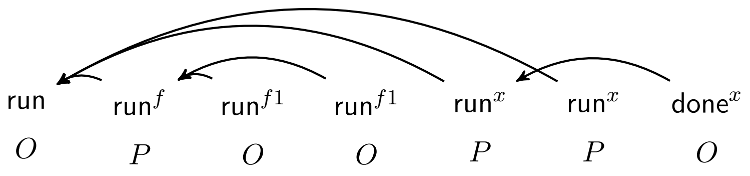


Figure 7.7: A play over arena A_2. The player who makes each move is denoted below the move.

Example 7.7. Consider the term $T = f : \mathbf{com} \to \mathbf{com}, x : \mathbf{com} \vdash fx$. Then we can construct the LA $\mathcal{A}_T$ corresponding to T as follows.

$\mathcal{A}_T = \langle Q, 3, \sigma, \delta \rangle$, where:

- $Q^{(0)} = \{0, 1, 2\}$, $Q^{(1)} = \{0\}$, $Q^{(2)} = \{0, 1, 2\}$, and $Q^{(3)} = \{0\}$;
- $\Sigma_{\mathsf{Q}} = \{\mathsf{run}, \mathsf{run}^f, \mathsf{run}^{f1}, \mathsf{run}^{(x,2)}\}$ and $\Sigma_{\mathsf{A}} = \{\mathsf{done}, \mathsf{done}^f, \mathsf{done}^{f1}, \mathsf{done}^x\}$; and
- δ is defined by the following transitions:

$$\dagger \xrightarrow{\mathsf{run}} 0 \qquad 0 \xrightarrow{\mathsf{run}^f} (1,0) \qquad (1,0) \xrightarrow{\mathsf{done}^f} 2 \qquad 2 \xrightarrow{\mathsf{done}} \dagger$$

$$(1,0) \xrightarrow{\mathsf{run}^{f1}} (1,0,0) \qquad (1,0,0) \xrightarrow{\mathsf{run}^{(x,2)}} (1,0,1,0)$$

$$(1,0,1,0) \xrightarrow{\mathsf{done}^{(x,0)}} (1,0,2) \qquad (1,0,2) \xrightarrow{\mathsf{done}^{f1}} (1,0)$$

The traces of $\mathcal{A}_T$ are all the plays in the strategy $\sigma = [\![T]\!]$, including the play given in Example 7.6. Moreover, the language $L(\mathcal{A}_T)$ represents $\mathsf{comp}(\sigma)$.

Example 7.8. It may be desirable to reduce the depth of the automaton given in Example 7.7. For example, we may imagine that the following data subtree is sufficient:

$$\begin{tikzcd}
& d_0 \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
d_1 \arrow[d, no head] \arrow[dr, no head] & d'_1 & d''_1 \\
d_2 & d'_2 &
\end{tikzcd}$$

The play from Example 7.6 would then be of the form

$$(\mathsf{run}, d_0)\,(\mathsf{run}^f, d_1)\,(\mathsf{run}^{f1}, d_2)\,(\mathsf{run}^{f1}, d'_2)\,(\mathsf{run}^x, d'_1)\,(\mathsf{run}^x, d''_1)\,(\mathsf{done}^x, d'_1)$$

hence "lifting" the interpretation of x to the top level. However, this cannot be done. Each instantiation of run^{f1} will enable exactly one run^x. So to be faithful to the set of complete plays, the automaton must track the number of instantiations of each to confirm that they are the same. Given the isolated nature of the branches in the LA configuration tree, the only point of synchronisation would be the root; which cannot be used to store the number of instantiations, since it is potentially unbounded and only finite amounts of information may be stored on nodes in LA.

We note some structural properties of the automata construction defined above. Recall (from Section 6.3.1) that reading a question letter always creates a leaf, and reading an answer letter always deletes one. The translation and iterative construction of the alphabet is such that letters corresponding to P-moves will always add leaves at odd levels (questions) and remove leaves at even levels (answers); while those corresponding to O-moves will add leaves at even levels and remove leaves at odd levels. Lastly, the automata construction fulfils an unusual property which we call *even-readiness*: we can structurally guarantee that, at the point where an even-level node is attempted to be removed, it will be a leaf.

This property of *even-readiness* is a consequence of the game-semantic WAIT condition (Definition 7.3). The condition captures the requirement that concurrent interactions be well-nested, i.e. that all child computations must terminate before their parent must terminate. In the case where data letters correspond to P-answers, it turns out this property always occurs. Formally, if the automaton arrives at configuration $\kappa = (D, E, f)$ then for any even level $2i$ data value $d \in E$, if there exists some transition

$$f(pred^{2i}(d), \cdots, pred(d), d) \xrightarrow{t} f'(pred^{2i}(d), \cdots, pred(d)) \in \delta_{\mathsf{A}}^{(2i)}$$

then d is guaranteed be a leaf. This is because the evolution of such even-level nodes is structural based on the term; the node will only reach a state of finality once the corresponding sub-computations are known to have completed. The same property does not hold for O-answers: the leafy automaton *must* check that odd-level nodes it is tasked with removing are leaves.

Theorem 7.2. *For any* FICA*-term $T = \Gamma \vdash M : \theta$, there exists an even-ready leafy automaton $\mathcal{A}_T$ over a finite subset of $\mathcal{T}_{\Gamma \vdash \theta} + \{\epsilon_{\mathsf{Q}}, \epsilon_{\mathsf{A}}\}$ such that the set of plays represented by traces of $\mathcal{A}_T$ is exactly $[\![\Gamma \vdash M : \theta]\!]$. Moreover, the language $L(\mathcal{A}_T)$ of complete traces of $\mathcal{A}_T$ represents $\mathsf{comp}([\![\Gamma \vdash M : \theta]\!])$ in the same sense.*

Proof. We present an inductive construction of a leafy automaton based on syntactic composition of normal-form FICA terms.

The base-case constructions are given here. As in Chapter 6, when giving an automaton we shall prescribe the transition relation of that automaton; all other parameters (finite alphabet, depth parameter, and set of states) are taken to be minimal with respect to the set of given transitions.

Where similarly named states appear at different levels, they are distinguished by their level to maintain the disjoint union of state levels. We shall use inference lines (——) to indicate that for every possible instantiation of the variables above the line meeting the given conditions, a corresponding transition should be created in the automaton of the shape below the line. At each step we are building a new automaton such that the antecedent of the rule (above the line) is replaced by the consequent (below the line). Where rules combine automata from multiple subterms, we shall use $\longrightarrow_i$ to indicate which of the subterms M_i the antecedent is derived from.

$\Gamma \vdash$ **skip** : **com**

$$\dagger \xrightarrow{\mathsf{run}} 0 \qquad 0 \xrightarrow{\mathsf{done}} \dagger$$

$\Gamma \vdash$ **div**$_{\mathbf{com}}$: **com**

$$\dagger \xrightarrow{\mathsf{run}} 0$$

$\Gamma \vdash$ **div**$_\theta$: θ

$$\frac{\theta = \theta_l \to \cdots \to \theta_1 \to \beta \qquad m \in M_{[\![\beta]\!]} \text{ is a question move}}{\dagger \xrightarrow{m} 0}$$

$\Gamma \vdash i :$ **exp**

$$\dagger \xrightarrow{\mathsf{q}} 0 \qquad 0 \xrightarrow{i} \dagger$$

Figure 7.8: Base cases for the construction of an LA from a FICA term.

Even-readiness is immediate in the base constructions: no configuration of any of the four LAs may have any node other than the root (since they are all 0-LAs), and so the root is always a leaf.

The remaining cases are inductive, and cover all remaining term-level constructs shown in the typing judgments of Figure 7.1. In all cases, take m to be from $\mathcal{T}_{\Gamma \vdash \theta} + \{\epsilon_{\mathsf{Q}}, \epsilon_{\mathsf{A}}\}$, and j from $\{-1, \ldots, k\}$; recall also that a move m listed without a superscript is shorthand for $m^{(\epsilon,0)}$. In some cases, different constructions will result in machines of different depths. Without loss of generality we shall assume that all constructions result in automata of the same depth; a k-LA can be viewed as a $(k+n)$-LA where the lower n levels are uninhabited.

$$\Gamma \vdash \mathbf{op}(M_1) : \mathbf{exp}$$

Applying a unary operation to an expression requires only minimal change. The computation for an expression terminates with a move returning the final value of that expression. We alter the transition to apply **op** to that returned value. The set of states is unchanged ($Q^{(j)} = Q_1^{(j)}$).

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \neq i}{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)}, \cdots, r_1^{(j')})} \qquad \frac{q_1^{(0)} \xrightarrow{i}_1 \dagger}{q_1^{(0)} \xrightarrow{\widehat{\mathbf{op}}(i)} \dagger}$$

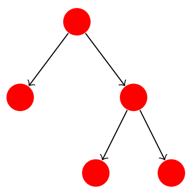
In the above, observe the special case where $j = -1$, and hence the sequence $(q_1^{(0)}, \cdots, q_1^{(j)})$ is empty (likewise for j'). Recall from Section 6.3.1 that the empty sequence in these cases is semantically the same as $\dagger$. Hence the general transition on the left covers all cases including the initial and final transitions, except for the single transition labelled by i. Even-readiness is obviously maintained by this construction, because neither the configuration graph nor the shape of any transition has changed.

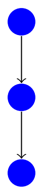
$$\Gamma \vdash M_1 || M_2 : \mathbf{com}$$

When performing parallel composition, we take the configuration graphs of our automata for M_1 and M_2 and "join" them at the root. The joining procedure is as follows: The root becomes the product of the states from the respective roots; at lower levels, we take only the disjoint sum of the respective states.

$$Q^{(0)} = Q_1^{(0)} \times Q_2^{(0)} \qquad\qquad Q^{(j)} = Q_1^{(j)} + Q_2^{(j)} \quad (1 \leq j \leq k)$$

We may imagine that the evaluation of the terms in parallel is performed in a “split-brain” fashion: transitions will modify children belonging only to the original M_1 part, and the corresponding part of the root; or they will modify children belonging to the original M_2 part, and the corresponding part of the root. As branches are isolated from each other, and the parallel-composed terms cannot interfere with each other below the root (by construction), every subtree rooted at a level-1 node will belong exclusively to either the M_1 part or the M_2 part.


(a) Some configuration of the automaton for M_1.


(b) Some configuration of the automaton for M_2.

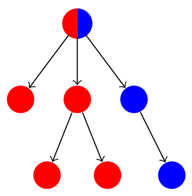

(c) The same configurations from (a) and (b), unified by parallel composition in M.

Figure 7.9: Sketch of the relationship of tree configurations inside the automata for M_1, M_2 and M.

The following transitions will activate and terminate the two components respectively. Note that this enforces that both sub-computations must have reached their own final states in order to delete the composite root node.

$$\frac{\dagger \xrightarrow{\mathsf{run}}_1 q_1^{(0)} \qquad \dagger \xrightarrow{\mathsf{run}}_2 q_2^{(0)}}{\dagger \xrightarrow{\mathsf{run}} (q_1^{(0)}, q_2^{(0)})} \qquad\qquad \frac{q_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger \qquad q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger}{(q_1^{(0)}, q_2^{(0)}) \xrightarrow{\mathsf{done}} \dagger}$$

The following transitions allow the automaton to make progress on one subcomputation or the other.

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad q_2^{(0)} \in Q_2^{(0)} \qquad \mathsf{m} \neq \mathsf{run}, \mathsf{done}}{((q_1^{(0)}, q_2^{(0)}), \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} ((r_1^{(0)}, q_2^{(0)}), \cdots, r_1^{(j')})}$$

$$\frac{q_1^{(0)} \in Q_1^{(0)} \qquad (q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \neq \mathsf{run}, \mathsf{done}}{((q_1^{(0)}, q_2^{(0)}), \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} ((q_1^{(0)}, r_2^{(0)}), \cdots, r_2^{(j')})}$$

Here, when we wish to make progress in the transitions from our automaton for M_1, we treat the component of the root state corresponding to M_2 as arbitrary, by making a new transition for each possible value of that part of the state). We then modify the component corresponding to M_1 as well as the branch which is known to originate from M_1 (as states from M_1 will be uniquely identifiable). The same logic applies symmetrically for updating M_2.

Even-readiness at level 2 and later follows directly from even-readiness for the subterms' automata, since the evolution of those subtrees is not changed by this construction. At level 0, the construction of done is such that the transition can only be fired if done could have been fired in the automata for both M_1 *and* M_2; by their own even-readiness they have no children at that point, and so the root of our composite automaton must also have no children. So this automaton is even-ready.

$$\Gamma \vdash M_1; M_2 : \mathbf{com}$$

This construction is a special case of the next, which covers $\Gamma \vdash M_1; M_2 : \beta$ for all base types β. We present this case separately to help illuminate the main points of the construction.

Recall that the sequential composition operator ; requires $M_1 : \mathbf{com}$. Given the two automata for M_1 and M_2, we shall compose the two such that all the transitions of M_1 will run except for the final done; followed by all the transitions of M_2 except for the initial run. This is mostly automatic: since states from the automata are taken to be disjoint, transitions from each automaton will never refer to states that appear in the other. We need only ensure that the handover from the first automaton to the second is performed appropriately.

The states of our new automaton are the union of those from the two sub-automata:

$$Q^{(i)} = Q_1^{(i)} + Q_2^{(i)} \qquad (0 \le i \le k).$$

Running the first automaton requires no changes beyond removing the done transition:

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{done}}{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)}, \cdots, r_1^{(j')})}$$

and running the second automaton requires no changes beyond removing the run transition:

$$\frac{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \neq \mathsf{run}}{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}.$$

So it only remains to manage the handoff from the first to the second. Whenever the M_1 automaton *could* have finished, we replace the transition to † to a transition to some first move (after initialising the root) in M_2.

$$\frac{q_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger \qquad \dagger \xrightarrow{\mathsf{run}}_2 q_2^{(0)} \qquad q_2^{(0)} \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \neq \mathsf{run}}{q_1^{(0)} \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

In the above construction, j' will always be either 1 or -1 for a well-formed LA: the first question move after adding the root may only either add a level-1 node, or delete the root again (with done).

This construction makes heavy use of the even-readiness property, and in particular the even-readiness of the automaton for M_1. As we move into the automaton for M_2, there will no longer be any transitions that delete nodes in states originating in M_1. So any level-1 children remaining from M_1 would no longer be removable, and the automaton would not be able to terminate. Thanks to even-readiness, we know that when the M_1 automaton could fire its done transition, its root was a leaf. As these are exactly the situations in which we move to M_2, the root must be a leaf when we move to M_2. The even-readiness of the M_2 automaton guarantees even-readiness of our composite automaton from that point onward, since we reuse M_2's transitions.

$$\Gamma \vdash M_1; M_2 : \beta$$

In the general case, we must track which initial move was played in order to identify the move to M_2 correctly. We use a product construction with the states from M_1 at the root in order to track the initial move that was made, along with unmodified M_2 states; and as in the **com** case, we take the disjoint sum of states at lower levels.

$$Q^{(0)} = (Q_1^{(0)} \times I) + Q_2^{(0)} \qquad\qquad Q^{(i)} = Q_1^{(i)} + Q_2^{(i)} \quad (1 \leq i \leq k)$$

We parameterise these constructions by the set I of initial moves, dependent on the base type β. (These are the same as the base moves listed in Section 7.2.)

The set I is defined as follows for each value of β.

β	I
com	$\{\mathsf{run}\}$
exp	$\{\mathsf{q}\}$
var	$\{\mathsf{read}\} + \{\mathsf{write}(i) \mid i \in \{0, \ldots, max\}\}$
sem	$\{\mathsf{grb}, \mathsf{rls}\}$

The transitions are as follows.

$$\frac{\dagger \xrightarrow{\mathsf{run}}_1 q_1^{(0)} \qquad x \in I}{\dagger \xrightarrow{x} (q_1^{(0)}, x)}$$

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{done} \qquad x \in I}{((q_1^{(0)}, x), \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} ((r_1^{(0)}, x), \cdots, r_1^{(j')})}$$

$$\frac{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \notin I}{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

$$\frac{q_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger \qquad \dagger \xrightarrow{x}_2 q_2^{(0)} \qquad q_2^{(0)} \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad x \in I \qquad \mathsf{m} \notin I}{(q_1^{(0)}, x) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

The first rule attaches the given move x at the top level; the second ensures that M_1 is interpreted correctly for all except the final done; the third ensures that M_2 is interpreted correctly for all except the initial run and the fourth hands off from M_1 to M_2, ensuring that the correct opening move is played. Even-readiness follows from M_1 and M_2 as in the **div** case described previously.

$$\Gamma \vdash \mathbf{newvar}\, x := i\, \mathbf{in}\, M_1 : \beta$$

Ghica and Murawski give in [70] a process by which we can derive $[\![\Gamma \vdash \mathbf{newvar}\, x := i\, \mathbf{in}\, M_1]\!]$ from $[\![\Gamma, x \vdash M_1]\!]$:

- firstly, restrict $[\![\Gamma, x \vdash M_1]\!]$ to only plays in which each $\mathsf{read}^x, \mathsf{write}(n)^x$ is immediately followed by its answer;
- restrict further to only those plays in which an answer to each read^x move is consistent with any preceding $\mathsf{write}(n)^x$ move (or i, if there is no such move);

- erase all moves relating to x, e.g. those of the form $m^{(x,\rho)}$.

This process serves to change the variable x from a free variable in $\Gamma, x \vdash M_1$ into a contained variable in $\Gamma \vdash \mathbf{newvar}\, x := i\, \mathbf{in}\, M_1$ which behaves coherently. To implement this procedure, we install a root-level *lock tag* such that the only valid move while the lock is in place is to play the answer move which removes it, after which point the automaton can continue freely. We will also maintain the current value of x at the root to ensure read moves receive the correct value, and write moves modify the value correctly. Eventually, all moves with the x subscript will be replaced with $\epsilon_{\mathsf{Q}}, \epsilon_{\mathsf{A}}$ to model hiding.

As described, our root-level state shall be a pair, where the first component is a level-0 state from M_1, with or without a *lock* tag, and the second component is the current value of the variable x.

$$Q^{(0)} = (Q_1^{(0)} + (Q_1^{(0)} \times \{lock\})) \times \{0, \ldots, max\} \qquad\qquad Q^{(j)} = Q_1^{(j)} \ \ (1 \le j \le k)$$

We initialise our stored value of x to i, and allow the computation to complete with x arbitrary:

$$\frac{\dagger \xrightarrow{\mathsf{m}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{m}} (q_1^{(0)}, i)} \qquad\qquad \frac{q_1^{(0)} \xrightarrow{\mathsf{m}}_1 \dagger \qquad 0 \le n \le max}{(q_1^{(0)}, n) \xrightarrow{\mathsf{m}} \dagger}.$$

We do not care about x for the sake of moves other than read^x, $\mathsf{write}(z)^x$, z^x and ok^x, so we merely lift such transitions from M_1 so that they preserve the current value of x stored at the root:

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \begin{array}{c} \mathsf{m} \ne \mathsf{read}^x, z^x, \mathsf{write}(z)^x, \mathsf{ok}^x \\ j, j' \ge 0 \qquad 0 \le n \le max \end{array}}{((q_1^{(0)}, n), \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} ((r_1^{(0)}, n), \cdots, r_1^{(j')})}.$$

When we read^x or $\mathsf{write}(z)^x$, we add a *lock* at level 0. This *lock* is removable only by a corresponding z^x or ok^x move—the lock guarantees that any prior reads/writes have already been answered, so only one move will be playable. Naturally, the values of reads and writes must be consistent with the value of x recorded at level 0. Observe that, in order to simulate hiding, we use ϵ_{Q} and ϵ_{A} rather than named letters here.

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{write}(z)^{(x,\rho)}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad 0 \le n, z \le max}{((q_1^{(0)}, n), \cdots, q_1^{(j)}) \xrightarrow{\epsilon_{\mathsf{Q}}} (((r_1^{(0)}, lock), z), \cdots, r_1^{(j')})}$$

$$\frac{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{read}^{(x,\rho)}}_1 (r_1^{(0)},\cdots,r_1^{(j')}) \qquad 0 \le n \le max}{((q_1^{(0)},n),\cdots,q_1^{(j)}) \xrightarrow{\epsilon_{\mathsf{Q}}} (((r_1^{(0)},lock),n),\cdots,r_1^{(j')})}$$

$$\frac{(r_1^{(0)},\cdots,r_1^{(j')}) \xrightarrow{\mathsf{ok}^x}_1 (t_1^{(0)},\cdots,t_1^{(j)}) \qquad 0 \le n \le max}{(((r_1^{(0)},lock),n),\cdots,r_1^{(j')}) \xrightarrow{\epsilon_{\mathsf{A}}} ((t_1^{(0)},n),\cdots,t_1^{(j)})}$$

$$\frac{(r_1^{(0)},\cdots,r_1^{(j')}) \xrightarrow{n^x}_1 (t_1^{(0)},\cdots,t_1^{(j)}) \qquad 0 \le n \le max}{(((r_1^{(0)},lock),n),\cdots,r_1^{(j')}) \xrightarrow{\epsilon_{\mathsf{A}}} ((t_1^{(0)},n),\cdots,t_1^{(j)})}$$

The basic operations of M_1 are not altered by this construction. Indeed, by the time we reach a finishing state from M_1 any addition or removal of nodes corresponding to these read^x and $\mathsf{write}(z)^x$ moves must necessarily have been cleaned up (due to the locking). So even-readiness will be maintained in our new automaton.

$$\Gamma \vdash \mathbf{newsem}\, s\, \mathbf{in}\, M_1 : \beta$$

This follows a similar pattern to $\mathbf{newvar}\, x := i\, \mathbf{in}\, M_1$. As `FICA` semaphores are binary, we restrict its root-stored value to $\{0,1\}$. We use the same locking mechanism as described in the $\mathbf{newvar}\, x := i\, \mathbf{in}\, M_1$ case to enforce that questions are immediately followed by their answers.

$$Q^{(0)} = (Q_1^{(0)} + (Q_1^{(0)} \times \{lock\})) \times \{0,1\} \qquad\qquad Q^{(j)} = Q_1^{(j)} \quad (1 \le j \le k)$$

We always initialise our semaphore as 0 to signify that it has not been grabbed by any computation. We allow computations to terminate while still in possession of the semaphore, so when finishing the automaton the value of the semaphore may be arbitrary.

$$\frac{\dagger \xrightarrow{\mathsf{m}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{m}} (q_1^{(0)},0)} \qquad \frac{q_1^{(0)} \xrightarrow{\mathsf{m}}_1 \dagger \qquad z \in \{0,1\}}{(q_1^{(0)},z) \xrightarrow{\mathsf{m}} \dagger}$$

Transitions which correspond to moves not concerning the semaphore (i.e. those other than $\mathsf{grb}^{(x,\rho)}$), $\mathsf{rls}^{(x,\rho)}$), and ok^x) proceed as before, taking the value of the semaphore to be arbitrary and preserving it through the transition.

$$\frac{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)},\cdots,r_1^{(j')}) \qquad z \in \{0,1\} \qquad \mathsf{m} \neq \mathsf{rls}^{(x,\rho)}, \mathsf{grb}^{(x,\rho)}, \mathsf{ok}^x}{((q_1^{(0)},z),\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}} ((r_1^{(0)},z),\cdots,r_1^{(j')})}$$

As with the **newvar** $x := i$ **in** M_1 case, we lock the automaton when we play the question moves for the semaphore ($\mathsf{grb}^{(x,\rho)}$ or $\mathsf{rls}^{(x,\rho)}$), removable only when reading the answer move (ok^x). Again we simulate hiding by the use of our pseudo-epsilon-transitions ϵ_Q and ϵ_A.

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{grb}^{(x,\rho)}}_1 (r_1^{(0)}, \cdots, r_1^{(j')})}{((q_1^{(0)}, 0), \cdots, q_1^{(j)}) \xrightarrow{\epsilon_\mathsf{Q}} (((r_1^{(0)}, lock), 1), \cdots, r_1^{(j')})}$$

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{rls}^{(x,\rho)}}_1 (r_1^{(0)}, \cdots, r_1^{(j')})}{((q_1^{(0)}, 1), \cdots, q_1^{(j)}) \xrightarrow{\epsilon_\mathsf{Q}} (((r_1^{(0)}, lock), 0), \cdots, r_1^{(j')})}$$

$$\frac{(r_1^{(0)}, \cdots, r_1^{(j')}) \xrightarrow{\mathsf{ok}^x}_1 (t_1^{(0)}, \cdots, t_1^{(j)}) \qquad z \in \{0,1\}}{(((r_1^{(0)}, lock), z), \cdots, r_1^{(j')}) \xrightarrow{\epsilon_\mathsf{A}} ((t_1^{(0)}, z), \cdots, t_1^{(j)})}$$

The same even-readiness argument from **newvar** works here too: thanks to the locking, any nodes introduced by $\mathsf{grb}^{(x,\rho)}$ and $\mathsf{rls}^{(x,\rho)}$ will be cleaned up in the very next move, and the automaton will never be in a finishing state in between. With all other moves being lifted from M_1 directly, even-readiness is maintained.

$$\Gamma \vdash f M_h \cdots M_1 : \mathbf{com} \qquad \text{where } f : \theta_h \to \cdots \to \theta_1 \to \mathbf{com} \in \Gamma$$

Observe that this also covers the case $f : \mathbf{com}$ (take $h = 0$).
In this construction, we will add two new levels at the "top" of the automaton for M_1. These will manage the computation spawning copies of $M_1, \ldots, M_h$. Our top-level state will have three values $\{0, 1, 2\}$. The level-1 state is meaningless and will not change; we arbitrarily choose 0. Beyond that, we take the (as always, disjoint) union of level-j states in the automata for $M_1, \ldots, M_h$ to be our states at level-$(j+2)$.

$$Q^{(0)} = \{0,1,2\} \qquad Q^{(1)} = \{0\} \qquad Q^{(j+2)} = \bigcup_{u \in \{1,\ldots,h\}} Q_u^{(j)} \quad (0 \le j \le k)$$

At the root, state 0 means "just created"; 2 means "about to finish"; and 1 means that f is currently being evaluated. State 1 is the only state in which the root shall have a child. We start with transitions corresponding to calling and returning from f:

$$\dagger \xrightarrow{\mathsf{run}} 0 \qquad 0 \xrightarrow{\mathsf{run}^f} (1,0) \qquad (1,0) \xrightarrow{\mathsf{done}^f} 2 \qquad 2 \xrightarrow{\mathsf{done}} \dagger\,.$$

Once we have the level-1 node in state 0, we want the environment to be able to spawn an unbounded number of copies of each of $\Gamma \vdash M_u : \theta_u$ (for $u \in \{1, \dots, h\}$). In what follows, note that we do not vary the level-0 or level-1 state; hence the transitions can be applied multiple times to summon many instances. Recall also that when j or j' are -1, the corresponding sequence is taken to be empty.

All moves originate either from some $M_u : \theta_u$, or from Γ.

- For moves $m^{(\vec{i},\rho)}$ from $M_u : \theta_u$, we annotate them further with f and with the index of the argument position u they appeared in. This allows us to easily distinguish which automaton the moves originated from.

$$\frac{(q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(\vec{i},\rho)}}_u (q_u^{(0)}, \cdots, q_u^{(j')})}{(1, 0, q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(fu\vec{i},\rho)}} (1, 0, q_u^{(0)}, \cdots, q_u^{(j')})}$$

  As pointers are depth-relative, moving all these transitions two layers deeper into the automaton will not change their meaning. We should adjust the initial moves from M_u so that it correctly points to run^f. As we use $\rho = 0$ to indicate the absence of a pointer, initial moves in the automata for $M_1 \dots M_h$ will already be set to 0, which as a non-initial move will point to the parent, which is run^f. So our pointer arithmetic is still correct without modifications.

- Moves from Γ are those of the form $m^{(x_v\vec{i},\rho)}$, where $1 \leq v \leq l$ and $(x_v : \theta_v) \in \Gamma$. We wish the question moves to be justified by run, which requires some adjustments to the value of ρ. To achieve this, we simply add 2 to ρ for question moves, and preserve ρ otherwise.

$$\frac{(q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(x_v,\rho)}}_u (q_u^{(0)}, \cdots, q_u^{(j')}) \qquad m \text{ is a question}}{(1, 0, q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(x_v,\rho+2)}} (1, 0, q_u^{(0)}, \cdots, q_u^{(j')})}$$

$$\frac{\left\{\begin{array}{c} (q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(x_v\vec{i},\rho)}}_u (q_u^{(0)}, \cdots, q_u^{(j')}) \\ \text{and} \\ \vec{i} \neq \epsilon \text{ or } (\vec{i} = \epsilon \text{ and } m \text{ is an answer}) \end{array}\right\}}{(1, 0, q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(x_v\vec{i},\rho)}} (1, 0, q_u^{(0)}, \cdots, q_u^{(j')})}$$

Even-readiness of this construction at level 0 is clear, since the root's singular child will be deleted by the transition into state 2, the only state from which the root itself may be deleted. All other even levels are simply (lowered) copies of the automata for $M_1 \dots M_h$, so even-readiness follows directly from those.

$$\Gamma \vdash fM_h \cdots M_1 : \mathbf{exp}$$

In the case where f returns a value i , we simply propagate i through the answers. The four transitions affecting levels 0 and 1:

$$\dagger \xrightarrow{\mathsf{run}} 0 \qquad 0 \xrightarrow{\mathsf{run}^f} (1,0) \qquad (1,0) \xrightarrow{\mathsf{done}^f} 2 \qquad 2 \xrightarrow{\mathsf{done}} \dagger$$

are replaced by these:

$$\dagger \xrightarrow{\mathsf{q}} 0 \qquad 0 \xrightarrow{\mathsf{q}^f} (1,0) \qquad (1,0) \xrightarrow{i^f} 2^i \qquad 2^i \xrightarrow{i} \dagger\,.$$

with all other constructions remaining the same.

$$\Gamma \vdash fM_h \cdots M_1 : \mathbf{var}$$

Terms of type **var** have two initial moves (read and $\mathsf{write}(x)$) and two corresponding final moves (i and ok). To distinguish between reads and writes we replace the root state 1 by two copies, 1^r and 1^w. Similarly we distinguish between 0 (reading) and 0^i (writing), and 2 (reading) and 2^i (writing). The opening and closing moves become these:

$$\dagger \xrightarrow{\mathsf{read}} 0 \qquad 0 \xrightarrow{\mathsf{read}^f} (1^r,0) \qquad (1^r,0) \xrightarrow{i^f} 2^i \qquad 2^i \xrightarrow{i} \dagger$$

$$\dagger \xrightarrow{\mathsf{write}(i)} 0^i \qquad 0^i \xrightarrow{\mathsf{write}(i)^f} (1^w,0) \qquad (1^w,0) \xrightarrow{\mathsf{ok}} 2 \qquad 2 \xrightarrow{\mathsf{ok}} \dagger \quad .$$

In addition, each of the transitions copied from M_u or Γ must now have two versions, where $(1,0,\cdots)$ is instead $(1^r,0,\cdots)$ and $(1^w,0,\cdots)$.

$$\Gamma \vdash fM_h \cdots M_1 : \mathbf{sem}$$

This case is similar to the **var** case. Here the questions are grb and rls, and the answer is always ok.

$$\dagger \xrightarrow{\mathsf{grb}} 0^g \qquad 0^g \xrightarrow{\mathsf{grb}^f} (1^g,0) \qquad (1^g,0) \xrightarrow{\mathsf{ok}^f} 2^g \qquad 2^g \xrightarrow{\mathsf{ok}} \dagger$$

$$\dagger \xrightarrow{\mathsf{rls}} 0^r \qquad 0^r \xrightarrow{\mathsf{rls}^f} (1^r,0) \qquad (1^r,0) \xrightarrow{\mathsf{ok}} 2^r \qquad 2^r \xrightarrow{\mathsf{ok}} \dagger \quad .$$

Like in the **var** case, each of the transitions copied from M_u or Γ must now have two versions, where $(1, 0, \cdots)$ is instead $(1^g, 0, \cdots)$ and $(1^r, 0, \cdots)$.

$$\Gamma \vdash \lambda x.M_1 : \theta_h \to \cdots \to \theta_1 \to \beta$$

Consider the automaton for $\Gamma, x : \theta_h \vdash M_1 : \theta_{h-1} \to \cdots \to \theta_1 \to \beta$. Our construction need only rename some labels in that automaton to achieve abstraction over x. In particular, transitions tagged with $m^{(x\vec{i},\rho)}$ should instead be tagged with $m^{(h\vec{i},\rho)}$, h being the new height of the term.

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{m^{(x\vec{i},\rho)}}_1 (r_1^{(0)}, \cdots, r_1^{(j)})}{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{m^{(h\vec{i},\rho)}} (r_1^{(0)}, \cdots, r_1^{(j)})}$$

$$\Gamma \vdash \mathbf{if}\, M_1\, \mathbf{then}\, M_2\, \mathbf{else}\, M_3 : \beta$$

This case is superficially similar to the $\Gamma \vdash M_1; M_2$ case: we must first run M_1, then hand control off to the second automaton. Here though M_1 has type **exp**, and its result dictates which of M_2 or M_3 will be run.

The states are as follows (recall the use of the initial move set I from the $\Gamma \vdash M_1; M_2$ case):

$$\begin{array}{rcll} Q^{(0)} & = & (Q_1^{(0)} \times I) + Q_2^{(0)} + Q_3^{(0)} & \\ Q^{(j)} & = & Q_1^{(j)} + Q_2^{(j)} + Q_3^{(j)} & \qquad (1 \le j \le k) \end{array}$$

We tag states from M_1 by x to denote which of the initial moves will be played by M_2 or M_3, so that the transition between them is smooth. We initialise x arbitrarily:

$$\frac{\dagger \xrightarrow{\mathsf{q}}_1 q_1^{(0)} \qquad x \in I}{\dagger \xrightarrow{x} (q_1^{(0)}, x)}.$$

Transitions in M_1 progress normally, albeit lifted to pass through the new x tag at the root:

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \notin \{0, \cdots, max\} \qquad x \in I}{((q_1^{(0)}, x), \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} ((r_1^{(0)}, x), \cdots, r_1^{(j')})}.$$

Moves in M_2 and M_3 progress normally, beyond their initial move:

$$\frac{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \notin I}{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}.$$

$$\frac{(q_3^{(0)}, \cdots, q_3^{(j)}) \xrightarrow{\mathsf{m}}_3 (r_3^{(0)}, \cdots, r_3^{(j')}) \qquad \mathsf{m} \notin I}{(q_3^{(0)}, \cdots, q_3^{(j)}) \xrightarrow{\mathsf{m}} (r_3^{(0)}, \cdots, r_3^{(j')})}.$$

These final two transitions are those which guide whether to fire off the automaton for M_2 or M_3 based on the result of M_1. Only one transition or the other will be fired: observe that the former will be fireable only when M_1 returns with a nonzero value; and the latter only when M_1 returns with 0.

$$\frac{q_1^{(0)} \xrightarrow{i}_1 \dagger \quad i > 0 \quad \dagger \xrightarrow{x}_2 q_2^{(0)} \quad q_2^{(0)} \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \quad x \in I \quad \mathsf{m} \notin I}{(q_1^{(0)}, x) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

$$\frac{q_1^{(0)} \xrightarrow{0}_1 \dagger \qquad \dagger \xrightarrow{x}_3 q_3^{(0)} \qquad q_3^{(0)} \xrightarrow{\mathsf{m}}_3 (r_3^{(0)}, \cdots, r_3^{(j')}) \qquad x \in I \qquad \mathsf{m} \notin I}{(q_1^{(0)}, x) \xrightarrow{\mathsf{m}} (r_3^{(0)}, \cdots, r_3^{(j')})}$$

Even-readiness follows from the same logic used in the $\Gamma \vdash M_1; M_2$ case.

$\Gamma \vdash$ **while** M_1 **do** M_2 : **com**

The **while** case requires surprisingly little accounting - we take the states to be the unions of the states from M_1 and M_2:

$$Q^{(j)} = Q_1^{(j)} + Q_2^{(j)} \qquad (0 \le j \le k)$$

We start by evaluating M_1, and terminate only in the case where M_1 gives back a value of zero. All other moves from M_1 are allowed to continue as normal with no modification.

$$\frac{\dagger \xrightarrow{\mathsf{q}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} q_1^{(0)}} \qquad \frac{q_1^{(0)} \xrightarrow{0}_1 \dagger}{q_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

$$\frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \notin \{\mathsf{q}, 0, \cdots, max\}}{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)}, \cdots, r_1^{(j')})}$$

When M_1 returns a nonzero value, we run M_2. Where we would initialise the automaton for M_2 we skip straight to the first move. We know, because M_2 : **com**, that the automaton for M_2 will start with run and end with done.

So instead we jump straight to the second move.

$$\frac{q_1^{(0)} \xrightarrow{i}_1 \dagger \qquad i > 0 \qquad \dagger \xrightarrow{\mathsf{run}}_2 q_2^{(0)} \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, r_2^{(1)}) \qquad \mathsf{m} \neq \mathsf{done}}{q_1^{(0)} \xrightarrow{\mathsf{m}} (r_2^{(0)}, r_2^{(1)})}$$

Sometimes there will be no such second move. We must handle this case specially, such that we skip straight back to evaluating M_1 again.

$$\frac{\left\{\begin{array}{c} q_1^{(0)} \xrightarrow{i}_1 \dagger \qquad i > 0 \qquad \dagger \xrightarrow{\mathsf{run}}_2 q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \\ \text{and} \\ \dagger \xrightarrow{\mathsf{q}}_1 r_1^{(0)} \xrightarrow{\mathsf{m}}_1 (u_1^{(0)}, u_1^{(1)}) \qquad \mathsf{m} \notin \{0, \dots, max\} \end{array}\right\}}{q_1^{(0)} \xrightarrow{\mathsf{m}} (u_1^{(0)}, u_1^{(1)})}$$

The automaton for M_2 progresses simply.

$$\frac{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \notin \{\mathsf{run}, \mathsf{done}\}}{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

When M_2 is finished, instead of using its own done transitions we instead reinitialise the automaton for M_1, completing the loop.

$$\frac{q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{q}}_1 q_1^{(0)} \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, r_1^{(1)}) \qquad \mathsf{m} \notin \{0, \dots, max\}}{q_2^{(0)} \xrightarrow{\mathsf{m}} (r_1^{(0)}, r_1^{(1)})}$$

Completing the loop in this way isn't possible when M_1 is trivial (i.e. summons no child nodes). In such cases, we are permitted to immediately terminate (when $i = 0$), or immediately return to M_2 (when $i > 0$).

$$\frac{q_1^{(0)} \xrightarrow{i}_1 \dagger \qquad i > 0 \qquad \dagger \xrightarrow{\mathsf{run}}_2 q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{q}}_1 r_1^{(0)} \xrightarrow{0}_1 \dagger}{q_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

$$\frac{q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{q}}_1 q_1^{(0)} \xrightarrow{0}_1 \dagger}{q_2^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

$$\frac{\left\{\begin{array}{c} q_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{q}}_1 q_1^{(0)} \xrightarrow{i}_1 \dagger \qquad i > 0 \\ \text{and} \\ \dagger \xrightarrow{\mathsf{run}}_2 r_2^{(0)} \xrightarrow{\mathsf{m}}_2 (u_2^{(0)}, u_2^{(1)}) \qquad \mathsf{m} \neq \mathsf{done} \end{array}\right\}}{q_2^{(0)} \xrightarrow{\mathsf{m}} (u_2^{(0)}, u_2^{(1)})}$$

As with other cases where we run automata sequentially (with minimal glue), even-readiness is inherited. No pointers need adjusting.

$$\Gamma \vdash !M_1 : \mathbf{exp}$$

When dereferencing M_1, M_1 must be of type **var**. To model dereferencing we need only replace instances of read tags with q tags. The set of states is unchanged: $Q^{(j)} = Q_1^{(j)}$ $(0 \leq j \leq k)$.

Transitions relevant to dereferencing M_1 are copied from the automaton for M_1 as follows.

$$\frac{\dagger \xrightarrow{\mathsf{read}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{q}} q_1^{(0)}} \qquad \frac{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{read}, \mathsf{write}(i), \mathsf{ok}}{(q_1^{(0)}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)}, \cdots, r_1^{(j')})}$$

With no changes to the structure of transitions or states, even-readiness follows immediately from M_1.

$$\Gamma \vdash M_1 := M_2 : \mathbf{com}$$

For assignment, we start by running M_2; the final move i is the value to assign to M_1, and we shall treat it as if $\mathsf{write}(i)$ was played. Once again we will take the unions of the states from the automata for M_1 and M_2:

$$Q^{(j)} = Q_1^{(j)} + Q_2^{(j)} \qquad (0 \leq j \leq k).$$

We start with M_2:

$$\frac{\dagger \xrightarrow{\mathsf{q}}_2 q_2^{(0)}}{\dagger \xrightarrow{\mathsf{run}} q_2^{(0)}} \qquad \frac{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}}{}_2 (r_2^{(0)}, \cdots, r_2^{(j')}) \qquad \mathsf{m} \notin \{0, \ldots, max\}}{(q_2^{(0)}, \cdots, q_2^{(j)}) \xrightarrow{\mathsf{m}} (r_2^{(0)}, \cdots, r_2^{(j')})}$$

Once M_2 is completed (i is ready to be fired), we start off the automaton for M_1 as if $\mathsf{write}(i)$ was played. Recall (again) that $(r_1^{(0)}, \cdots, r_1^{(j')})$ may be $\dagger$.

$$\frac{\left\{ \begin{array}{c} q_2^{(0)} \xrightarrow{i}_2 \dagger \qquad i \in \{0, \cdots, max\} \qquad \dagger \xrightarrow{\mathsf{write}(i)}_1 q_1^{(0)} \\ \text{and} \\ q_1^{(0)} \xrightarrow{\mathsf{m}}_1 (r_1^{(0)}, \cdots, r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{ok} \end{array} \right\}}{q_2^{(0)} \xrightarrow{\mathsf{m}} (r_1^{(0)}, \cdots, r_1^{(j')})}$$

The automaton for M_1 now progresses, but further reads or writes to/from

M_1 are ignored.

$$\frac{\left\{\begin{array}{c}(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)},\cdots,r_1^{(j')}) \\ \text{and} \\ \mathsf{m} \notin \{\mathsf{read},\mathsf{write}(0),\ldots,\mathsf{write}(max),0,\ldots,max,\mathsf{ok}\}\end{array}\right\}}{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)},\cdots,r_1^{(j')})}$$

Finally, M_1 is allowed to complete.

$$\frac{q_1^{(0)} \xrightarrow{\mathsf{ok}}_1 \dagger}{q_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

No pointer adjustments are needed here, and again even-readiness follows directly from even-readiness in the automata for M_1 and M_2.

$\Gamma \vdash \textbf{grab}(M_1) : \textbf{com}$ and $\Gamma \vdash \textbf{release}(M_1) : \textbf{com}$

These two cases are the same. We simply rename the initial instance of grb (respectively, rls) to run, and the final ok to done. Even-readiness is obviously maintained, as correctness of pointers.

For **grab**(M_1):

$$\frac{\dagger \xrightarrow{\mathsf{grb}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} q_1^{(0)}} \qquad \frac{q_1^{(0)} \xrightarrow{\mathsf{ok}}_1 \dagger}{q_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

$$\frac{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)},\cdots,r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{grb},\mathsf{rls},\mathsf{ok}}{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)},\cdots,r_1^{(j')})}.$$

For **release**(M_1):

$$\frac{\dagger \xrightarrow{\mathsf{rls}}_1 q_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} q_1^{(0)}} \qquad \frac{q_1^{(0)} \xrightarrow{\mathsf{ok}}_1 \dagger}{q_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

$$\frac{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}}_1 (r_1^{(0)},\cdots,r_1^{(j')}) \qquad \mathsf{m} \neq \mathsf{grb},\mathsf{rls},\mathsf{ok}}{(q_1^{(0)},\cdots,q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(0)},\cdots,r_1^{(j')})}.$$

□

The above shows that leafy automata are capable of faithfully modelling the game semantics of `FICA` terms, such that a term's plays can be exactly represented in the traces of a constructed automaton, and its language comprises the complete plays. This encoding alone is not enough to give us any decidability results, of course. However, restricting or altering the structure of our leafy automaton may be a fruitful new mechanism through which to discover corresponding properties or restrictions of the language. In the following chapters we shall explore that approach in a number of different ways.

Chapter 8

Local Leafy Automata

In Chapter 6, we relayed that many problems on general leafy automata are undecidable. This naturally poses some difficulties when trying to solve instances of verification problems via LA. Across the next two chapters, we shall expound two new varieties of leafy automata: firstly *local leafy automata* in this chapter; and then *split automata* in Chapter 9. In both cases, we identify a fragment of the FICA language which can be translated to the restricted automaton variant, and show that certain problems over that fragment are decidable by reduction to their respective variant.

8.1 Boundedness

In order to derive value from the local model described here, we must first discuss a notion of *boundedness*. Boundedness is a property of an individual automaton—it is not structurally guaranteed by any of the models we shall discuss, and does not form part of the definition of those models. The property can be inferred from an automaton based on its transitions taken as a whole; when making use of boundedness for any given automaton we shall first show that it holds.

Definition 8.1. A leafy automaton $\mathcal{A}$ (be that a k-LA or any variant thereof) is *bounded at level i* if and only if there is some $b \in \mathbb{N}$ such that each node at level i may create at most b children during any run. The maximum value of b across all bounded levels in any automaton $\mathcal{A}$ with at least one bounded level is called the *branching bound*.

For the remainder of this work, we shall assume that for any automaton with at least one bounded level, all such levels share the same bound, simply called b. (Over-estimating the branching bound will not cause any issues, and the proofs are made simpler as a result.)

Definition 8.2. An automaton is *even-bounded* if it is bounded at every even level $2i$ for $i \in \mathbb{N}$.

The above definition will be convenient for our proofs; later we shall see that this property naturally occurs in certain local leafy automaton constructions.

8.2 Local Leafy Automata

Local leafy automata (LLA) are a restriction of the general leafy automata from Chapter 6. We shall therefore borrow much of the notation, semantics and terminology described in that chapter when discussing local leafy automata, and LLA shall primarily be discussed in terms of the differences between them and LA.

Definition 8.3. A level-k *local* leafy automaton (k-LLA) is a quadruple $\mathcal{A} = \langle \Sigma, k, Q, \delta \rangle$, where

- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is a finite alphabet, partitioned into questions and answers;
- $k \geq 0$ is the level parameter;
- $Q = \sum_{i=0}^{k} Q^{(i)}$ is a finite set of states, partitioned into sets $Q^{(i)}$ of level-i states;
- $\delta = \delta_{\mathsf{Q}} + \delta_{\mathsf{A}}$ is a finite transition relation, partitioned into question- and answer-related transitions;
- $\delta_{\mathsf{Q}} = \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \delta_{\mathsf{Q}}^{(2i)} + \delta_{\mathsf{Q}}^{(2i+1)}$ are the question transitions, paritioned into odd and even layers such that
  - $\delta_{\mathsf{Q}}^{(2i)} \subseteq Q^{[2i-2,\ 2i-1]} \times \Sigma_{\mathsf{Q}} \times Q^{[2i-2,\ 2i]}$
  - $\delta_{\mathsf{Q}}^{(2i+1)} \subseteq Q^{[2i-2,\ 2i]} \times \Sigma_{\mathsf{Q}} \times Q^{[2i-2,\ 2i+1]}$
- $\delta_{\mathsf{A}} = \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \delta_{\mathsf{A}}^{(2i)} + \delta_{\mathsf{A}}^{(2i+1)}$ are the answer transitions, partitoned into odd and even layers such that
  - $\delta_{\mathsf{A}}^{(2i)} \subseteq Q^{[2i-2,\ 2i]} \times \Sigma_{\mathsf{A}} \times Q^{[2i-2,\ 2i-1]}$
  - $\delta_{\mathsf{A}}^{(2i+1)} \subseteq Q^{[2i-2,\ 2i+1]} \times \Sigma_{\mathsf{A}} \times Q^{[2i-2,\ 2i]}$

We shall assume, with no loss of generality[1], that there are always an even number of levels in any LLA.

[1]To treat a k-LLA as a $(k+1)$-LLA, fix $Q^{(k+1)} = \emptyset$. No further changes are required.

The definition of LLA is, in many respects, the same as that of LA. The critical distinction lies in the transition function δ. There are two important differences between the transition function as described in LA and that in LLA.

Foremost, the automaton no longer has the ability to define transitions with respect to the states along an entire branch from root to leaf when adding or removing a leaf. Instead, the scope of observations up the tree is limited to only two or three levels.

Secondly we distinguish between whether a level is *odd* or *even* (that is, whether the level number is $2i$ or $2i + 1$ for some integer i). We do this so that odd levels are permitted access to one additional level up the tree when adding or removing states according to the transition function.

The two changes above can be briefly summed up as follows: The addition or removal of any node, whether at an odd or even level, may not read nor write further than the nearest even level's grandparent. So additions/removals at level $2i$ or $2i + 1$ may make changes as far as level $2i - 2$, but no further. Of course, the restriction still applies that these changes must take place only on the path between the newly added/removed node and the root. This is where the "local" in "local leafy automata" comes from: transitions adding a leaf may only make use of the local part of the branch to which said leaf is being added. The semantic effect of this change will be described later in this chapter.

It may be surprising to learn that this change is already sufficient to make certain problems decidable.

Theorem 8.1. *The emptiness problem for even-bounded k-LLA is decidable.*

Proof. Let us set the scene with a quick sketch of the proof, before giving the full working.

Fix b as the branching bound of our even-bounded k-LLA. The main observation is this: once a node d at even level $2i$ has been created, all subsequent actions of descendents of d access (that is, read and/or write) the states at levels $2i - 1$ and $2i - 2$ at most $2b$ times. The shape of the transition function dictates that this can happen only when children of d at level $2i + 1$ are added or removed. Similarly, it is not possible for any node at level $< (2i - 2)$ to be accessed at the same time as d or any of its descendents.

We make use of this observation to construct *summaries* of nodes on even levels which completely describe all possible lifetimes of nodes at that level, from their creation until their removal, and in between performing at most $2b$ read-writes of the parent and grandparent states. A summary (formally described below) is a complete record of a valid sequence of read-writes and the corresponding stateful changes for any node at level $2i$.

We shall prove by induction that, given the summaries for nodes at level $2i+2$, we can construct the summaries for nodes at level $2i$. We will construct a program for testing whether some set of the correct shape is a valid summary at level $2i$, based on the summaries at level $2i+2$. The program will be written in a simple language over infinite counters, which can then be executed on VASS. Since there are finitely many possible summaries at any (even) level, we can enumerate them all and hence compute all summaries at all such levels in decreasing order. We proceed until level 2, at which point we can reduce to reachability on a VASS.

The complete proof follows.

* * *

We split the proof into two parts. The first part induces a notion of *summaries*, which describe all valid sequences of possible states and reads-writes that a node at a given level may perform. Secondly, we use that summary notion to give an algorithm for generating summaries at level $2i$ based on summaries at level $2i+2$, which then powers the inductive argument.

8.2.1 Summaries

The structure of k-LLA transitions gives rise to a notion of a domain for data values. The *domain* of a data value $d \in \mathcal{D}$ is the set of data values whose associated state may be modified by a transition that adds or removes d (in other words, when reading a data letter which has d as its data value.) Recall the definition of *pred* from Section 6.2.

$$\mathsf{dom}(d) = \begin{cases} \{pred^2(d), pred(d), d\} & \text{if d is at an even level} \\ \{pred^3(d), pred^2(d), pred(d), d\} & \text{if d is at an odd level} \end{cases}$$

Observe that this definition is not well-formed for any data value d at level 0, 1, or 2. This case is treated separately at the end of the proof.

We shall also make use of a notion of independence which is based on these domains. Two data letters (t_1, d_1), (t_2, d_2) are said to be *independent* if the domains of d_1 and d_2 are disjoint. Independence is a strong condition with implications for the set of traces:

Lemma 8.1. *If w is a trace of some* LLA *$\mathcal{A}$, then every sequence obtained by permuting adjacent independent letters of w is also a trace of $\mathcal{A}$. Moreover the final configuration of all such traces is the same.*

Proof. Consider a trace $u(t_1, d_1)(t_2, d_2)$ such that (t_1, d_1) and (t_2, d_2) are independent, with u some arbitrary trace. Note that the domain of d_2 corresponds to those data values whose nodes' states may be read and/or written when adding or removing d_2 as a leaf. If d_1 is not in that domain, then the move adding or removing d_2 cannot be predicated on the existence of d_1 nor on any state modified by the addition or removal of d_1. Hence if $u(t_1, d_1)(t_2, d_2)$ is a trace then $u(t_2, d_2)$ is also a trace. Symmetrically, since d_2 is not in the domain of d_1, we can see that if $u(t_2, d_2)$ is a trace then $u(t_2, d_2)(t_1, d_1)$ must also be a trace. It follows that the states of all nodes after trace $u(t_1, d_1)(t_2, d_2)$ are the same as those after $u(t_2, d_2)(t_1, d_1)$.

We need not worry about the case where (say) d_2 is a descendent of d_1, but d_1 is not in the domain of d_2 because it's too far down the tree; such letters could never be adjacent in a trace as any intermediate nodes must be created/destroyed in between such occurrences. $\square$

Let us fix an even-bounded k-LLA $\mathcal{A} = \langle \Sigma_{\mathcal{A}}, k_{\mathcal{A}}, Q_{\mathcal{A}}, \delta_{\mathcal{A}} \rangle$, with branching bound b.

Suppose that, in an accepting trace on $\mathcal{A}$, we encounter some data value d at even layer $2i$. In any such trace, d must appear exactly twice. The first occurrence corresponds to adding d, and the second to deleting d. Let w be the part of the trace in between, and including, these two occurrences of d. We can classify each data letter (t', d') of w into one of three categories:

1. *d-internal*, when $\mathsf{dom}(d')$ is entirely within the subtree rooted at d;
2. *d-external*, when $\mathsf{dom}(d')$ is disjoint from the subtree rooted at d;
3. *d-frontier*, when $\mathsf{dom}(d')$ contains d and its parent.

These three classes partition the set of all data letters in w. The frontier letters are those with data values d and the children of d (the latter being from level $2i + 1$). Data letters with data values from levels $> 2i + 1$ are either d-internal or d-external.

Note that no child of d will appear in the trace outside of w, since (per Remarks 6.1 and 6.2) there will be an unanswered question associated with d any time a child of d is created or destroyed. Since d has even level, and $\mathcal{A}$

is even-bounded, the number of children that d gains across a run is bounded by b, and thus the total number of d-frontier letters in w is at most $b+1$.

We now split w into subwords, such that the boundaries of these subwords are the d-frontier letters. This creates a sequence of transitions:

$$\kappa_1 \xrightarrow{m_1} \kappa_1' \xrightarrow{w_1} \kappa_2 \xrightarrow{m_2} \kappa_2' \xrightarrow{w_2} \ldots \kappa_l \xrightarrow{m_l} \kappa_l' \xrightarrow{w_l} \kappa_{l+1} \xrightarrow{m_{l+1}} \kappa_{l+1}' \tag{8.1}$$

In Equation 8.1, $m_1, \ldots, m_l$ are the d-frontier data letters. The first and last are distinguished: m_1 adds the node annotated by d to the tree and m_{l+1} deletes it.

Configuration κ_1' is the first in which d appears in the tree, so d is a leaf node in κ_1'. Likewise, κ_{l+1} is the last configuration in which d appears, as it is removed by m_{l+1}, so (per Remark 6.2) d is a leaf node in κ_{l+1}.

We may now make use of the independence properties of the data letters in subwords. Every word w_j contains only d-internal and d-external data letters. Since d-internal and d-external data letters will necessarily be independent, we can safely rearrange w_j into two words $u_j \cdot v_j$ such that u_j is the subword of w_j consisting only of the d-internal letters, and v_j is the subword of w_j consisting only of the d-external letters. u_1 and u_l will be empty.

From here, we can see that the d-internal parts $u_1, \cdots, u_l$ of w only interact with the d-external parts $v_1, \cdots, v_l$ at a bounded number of positions, and those positions exactly correspond to the frontier transitions $m_2, \cdots, m_l$. Hence, if we could characterize the interactions that can occur at level $2i$, then we could replace the sequences of transitions on every u_j by some single short-cut transition. This would eliminate the need for levels $\geq 2i$ in the automaton.

We introduce the notion of a summary to implement such short-cut transitions. A *summary* for level $2i$ is a function

$$f : \{1, \ldots, 2(l+1)\} \longrightarrow Q^{2i-2} \times Q^{2i-1} \tag{8.2}$$

for some $l \leq b+1$.

Intuitively, from a trace w expanded as in Equation 8.1, we can extract f such that $f(2j-1)$ is a pair of states labelling $\mathit{pred}^2(d)$ and $\mathit{pred}(d)$ in κ_j, while $f(2j)$ is a pair of states labelling these nodes in κ_j'. Of course we do not construct f based on any particular runs of $\mathcal{A}$ in practice; instead we shall decide, for any given f, whether f is a summary based on the transition function of $\mathcal{A}$, using the process described below.

To formalise the idea of summaries for a given automaton, we will introduce the notion of a *cut automaton.* Intuitively, the behaviour of a cut automaton $\mathcal{A}^{\downarrow}(2i, f)$ will represent the behaviours of $\mathcal{A}$ contained within some subtree rooted in a data value at layer $2i$, and following some summary f. The information that is "cut" is all the states and transitions at levels $\{0 \dots 2i-1\}$.

The states and transitions of $\mathcal{A}^{\downarrow}(2i, f)$ are those of $\mathcal{A}$ but modified so that level $2i$ becomes the root level. Recall that, since these transitions are local (their reach is bounded), we do not need to modify any values within the transitions themselves.

$$\left.\begin{aligned} \mathsf{Q}^{\downarrow(l-2i)} &= \mathsf{Q}^{(l)} \\ \delta_{\mathsf{Q}}^{\downarrow(l-2i)} &= \delta_{\mathsf{Q}}^{(l)} \\ \delta_{\mathsf{A}}^{\downarrow(l-2i)} &= \delta_{\mathsf{A}}^{(l)} \end{aligned}\right\} \text{for } l \geq 2i+2$$

The two topmost layers, $2i \Rightarrow 0$ and $(2i+1) \Rightarrow 1$, must be treated specially since their parents and grandparents are "cut" by this procedure. Hence we use an adjusted annotation for those levels. We also annotate the state at the root by some integer from the domain of f, which shall be used by the transitions at layer 1.

$$Q^{\downarrow(0)} = Q^{(2i)} \times \mathsf{dom}(f)$$

$$Q^{\downarrow(1)} = Q^{(2i+1)}$$

Before defining the transitions of $\mathcal{A}^{\downarrow}(2i, f)$ we introduce some further notation. For a summary f we write $\mathsf{max}(\mathsf{dom}(f))$ for the maximal element in the domain of f. We shall also make use of an abbreviated notation for transitions. Recall that within a summary, $f(j)$ and $f(j+1)$ are definitionally linked (for odd j) such that $f(j)$ and $f(j+1)$ intuitively correspond to the values being "read" and "written" (respectively) by addition or removal of nodes in the d-frontier. So if $f(j) = (q^{(2i-2)}, q^{(2i-1)})$, and $f(j+1) = (q'^{(2i-2)}, q'^{(2i-1)})$, then we write

$$f(j) \xrightarrow{a} (f(j+1), q'^{(2i)})$$

instead of

$$(q^{(2i-2)}, q^{(2i-1)}) \xrightarrow{a} (q'^{(2i-2)}, q'^{(2i-1)}, q'^{(2i)}) \ .$$

The transitions of $\mathcal{A}^{\downarrow}(2i, f)$ at levels 0 and 1 are adaptations of those of levels $2i$ and $2i+1$ in $\mathcal{A}$. A node that was at level $2i$ is now the root, and so it has no predecessors. The initial and final moves of $\mathcal{A}^{\downarrow}(2i, f)$ will create and destroy the root. These transitions make use of f in order to create and destroy the root (i.e. the node labelled by d) in a way which is faithful to $\mathcal{A}$.

In the translation rules given below, the antecedent (above the line) is some transition that exists in $\mathcal{A}$, and the consequent (below the line) is the induced transition in $\mathcal{A}^{\downarrow}(2i, f)$.

$$\frac{f(1) \xrightarrow{a} (f(2), q'^{(2i)}) \in \delta_{\mathsf{Q}}^{(2i)}}{\dagger \xrightarrow{a} (q'^{(2i)}, 3) \in \delta_{\mathsf{Q}}^{\downarrow(0)}}$$

$$\frac{(f(r), q) \xrightarrow{a} f(r+1) \in \delta_{\mathsf{A}}^{(2i)}}{(q, r) \xrightarrow{a} \dagger \in \delta_{\mathsf{A}}^{\downarrow(0)}} \qquad \text{for } r = \mathsf{max}(\mathsf{dom}(f)) - 1$$

Lastly we have transitions that add and delete nodes at level 1:

$$\frac{(f(r), q^{(2i)}) \xrightarrow{a} (f(r+1), q'^{(2i)}, q'^{(2i+1)}) \in \delta_{\mathsf{Q}}^{(2i+1)}}{(q^{(2i)}, r) \xrightarrow{a} ((q'^{(2i)}, r+2), q'^{(2i+1)}) \in \delta_{\mathsf{Q}}^{\downarrow(1)}}$$

$$\frac{(f(r), q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} (f(r+1), q'^{(2i)}) \in \delta_{\mathsf{A}}^{(2i+1)}}{((q^{(2i)}, r), q^{(2i+1)}) \xrightarrow{a} ((q'^{(2i)}, r+2)) \in \delta_{\mathsf{A}}^{\downarrow(1)}}$$

We can now formally define the set of summaries for an even layer $2i$:

$$\mathit{Summary}(\mathcal{A}, 2i) = \{f \mid \mathcal{A}^{\downarrow}(2i, f) \text{ accepts some trace}\}$$

The next step is to define an automaton that uses such a set of summaries. The idea is that when a node at level $2i$ is created it is assigned a summary from the set of summaries. Then all moves below this node are simulated by consulting this summary, and hence we will never need to directly emulate the levels past $2i$.

Let $\mathcal{S}$ be a set of summaries at level $2i$. We will now define $\mathcal{A}^{\uparrow}(2i, \mathcal{S})$ as a $(2i+1)$-LLA. The states and transitions of $\mathcal{A}^{\uparrow}(2i, \mathcal{S})$ are exactly the states and transitions of $\mathcal{A}$ for levels $\{0, \cdots, 2i-1\}$. The set of states at level $2i$ is

$$Q^{(2i)} = \{(f, r) \colon f \in \mathcal{S}, r \in \mathsf{dom}(f)\} \ .$$

So a state at layer $2i$ is a summary function and a *use counter* indicating the part of the summary that has been discharged so far in a run.

For technical reasons[2] we will also need one state at layer $2i+1$. We set $Q^{(2i+1)} = \{\bullet\}$.

[2]Per the definition of LLA, changing the state at level $2i$ requires the addition or removal of a descendant. The child that is added in this way does not itself bear any meaning.

The transitions of $\delta_{\mathsf{Q}}^{\uparrow(2i)}$ and $\delta_{\mathsf{A}}^{\uparrow(2i)}$ are defined as follows.

$$f(1) \xrightarrow{a} (f(2), (f, 3)) \in \delta_{\mathsf{Q}}^{\uparrow(2i)} \qquad \text{for } f \in \mathcal{S}$$

$$(f(r), (f, r)) \xrightarrow{a} f(r+1) \in \delta_{\mathsf{A}}^{\uparrow(2i)} \qquad \text{for } r = \mathsf{max}(\mathsf{dom}(f)) - 1$$

These transitions imply that for every node created at level $2i$, the automaton nondeterministically guesses a summary and sets the summary's use counter to 3. (It is 3 and not 1 because the first two values of f are used for the creation of the node.) The node can be deleted once this bounded counter value is maximal—this corresponds to all children of the simulated node having been removed, and the state itself being in a removable condition as per the original definition of $\mathcal{A}$.

Finally, we define the transitions of $\delta_{\mathsf{Q}}^{\uparrow(2i+1)}$ and $\delta_{\mathsf{A}}^{\uparrow(2i+1)}$.

$$(f(r), (f, r)) \xrightarrow{a} (f(r+1), (f, r+2), \bullet) \in \delta_{\mathsf{Q}}^{\uparrow(2i+1)} \qquad \text{if } r < \mathsf{max}(\mathsf{dom}(f)) - 1$$

$$(f(r), (f, r), \bullet) \xrightarrow{a} (f(r), (f, r)) \in \delta_{\mathsf{A}}^{\uparrow(2i+1)} \qquad \text{if } r = \mathsf{max}(\mathsf{dom}(f)) - 1$$

Whenever the automaton follows one of the above question transitions it discharges another step in some node's summary and creates a trivial node ($\bullet$) as a child of that node. The use counter of that node is increased by 2 at such a transition. Once the use counter cannot be increased any more, $\delta_{\mathsf{A}}^{\uparrow(2i+1)}$ provides transitions for repeatedly deleting those trivial nodes. No other transitions are applicable at this point. Once there are no children, the simulated node can be removed by a $\delta_{\mathsf{A}}^{\uparrow(2i)}$ transition.

For illustrative purposes, consider some node n at level $2i$. Recall the meaning of the transitions whose first component is $f(\ldots)$: these are states of the parent and grandparent of the summarizing node n. When the summary-progressing transition fires, it does so on the condition that its grandparent and parent states are as dictated by $f(r)$ and the result is that the parent and grandparent states are set to $f(r+1)$. The node's summary is able to progress again when the states at the parent and grandparent are configured according to $f(r+2)$, and so on. If $f(r+1)$ and $f(r+2)$ coincide then this can be immediate; otherwise it can occur by some sibling of n changing the states of n's parent and grandparent in the same way.

The next lemma formally states the relationship between the two automata we have introduced (cut automata $\mathcal{A}^{\downarrow}$ and the simulating automata $\mathcal{A}^{\uparrow}$) and the original $\mathcal{A}$.

Lemma 8.2. *For every k-*LLA *$\mathcal{A}$ and level $2i < k$, $\mathcal{A}$ accepts a trace if and* *only if* $\mathcal{A}^{\uparrow}(2i, \mathit{Summary}(\mathcal{A}, 2i))$ *accepts a trace.*

Proof. We will first construct an accepting trace $w^{\uparrow}$ of $\mathcal{A}^{\uparrow}(2i, \mathit{Summary}(\mathcal{A}, 2i))$ based on an accepting trace w of $\mathcal{A}$.

Since w is accepting, for each data value d at level $2i$ that appears in w, d must appear exactly twice in w: once to add the node corresponding to d and then later to remove it. Hence the subsequence of w comprising data letters in the subtree of d must itself be an accepting trace of $\mathcal{A}^{\downarrow}(2i)$. The data letters of w can be partitioned into those at levels $< 2i$ and those at levels $\geq 2i$. The former are maintained in our new accepting trace. The latter may be further partitioned into some number of accepting traces of $\mathcal{A}^{\downarrow}(2i)$. Each such trace is rooted in some data value $d^{\downarrow}$ at level $2i$, and hence corresponds to some summary in $\mathcal{A}^{\uparrow}(2i, \mathit{Summary}(\mathcal{A}, 2i))$.

Letters in such traces are either $d^{\downarrow}$-internal or $d^{\downarrow}$-frontier. The $d^{\downarrow}$-frontier letters are maintained in $w^{\uparrow}$: they allow us to progress the summaries at level $2i$. For each d-frontier data letter in w we replace it by a pair of data letters which allow us to progress those summaries per the transitions of $\mathcal{A}^{\uparrow}(2i, \mathit{Summary}(\mathcal{A}, 2i))$. The $d^{\downarrow}$-internal letters are no longer required; the behaviours corresponding to those letters are exactly those being summarised.

We now show that if an accepting trace $w^{\uparrow}$ of $\mathcal{A}^{\uparrow}(2i, \mathit{Summary}(\mathcal{A}, 2i))$ exists then an accepting trace w of $\mathcal{A}$ must also exist. This is done by stitching runs of $\mathcal{A}^{\uparrow}$ and $\mathcal{A}^{\downarrow}$ and proceeds symmetrically to the previous argument.

If $w^{\uparrow}$ is accepting, then $w^{\uparrow}$ consists of data letters at levels $< 2i$, data letters at level $2i$, and some number of data letters at level $2i + 1$ which facilitate moving through summaries. By definition, each summary at level $2i$ corresponds to a sequence of additions/removals of nodes in the subtree rooted at some data value d at level $2i$, starting with the addition of d and ending with its removal. Hence each such summary must correspond to an accepting trace of $\mathcal{A}^{\downarrow}(2i)$. So the data letters of some accepting trace of $\mathcal{A}^{\downarrow}(2i)$ rooted at d (for each such d), taken together with the data letters at level $< 2i$ from w, in some interleaving, form an accepting trace of w. □

The next lemma declares that we can use summaries of level $2i + 2$ to compute summaries at level $2i$.

Lemma 8.3. *Take a summary f of some level $2i$, and consider $\mathcal{B} = \mathcal{A}^{\downarrow}(2i, f)$. Then $\mathcal{B}$ accepts some trace if and only if $\mathcal{B}^{\uparrow}(2, \mathit{Summary}(\mathcal{A}, 2i + 2))$ accepts some trace.*

Proof. Observe that $\mathit{Summary}(\mathcal{A}, 2i+2) = \mathit{Summary}(\mathcal{B}, 2)$, since the behaviours at level $2i+2$ in automaton $\mathcal{A}$ and level 2 in $\mathcal{B}$ are the same. The result follows from this and from Lemma 8.2. Observe that a summarising automaton with depth $2i+2$ is itself a $(2i+2)$-LLA. Hence, it can be itself summarised by a summarising automaton with depth $2i$. □

We repeatedly apply Lemma 8.3 to reduce the depth of the simulating automaton from $2k$ to depth at most 3. The task of computing summaries then reduces to checking emptiness of 3-LLA. In the second part of the proof we show how to reduce the latter problem to reachability in VASS. This turns out to be a degenerate case of computing summaries, so the same technique as for computing summaries is used.

We now proceed to explain the summary computation procedure.

Computing summaries

Computation of summaries relies on the following fact:

Lemma 8.4. *For any i, the number of summaries at level $2i$ is bounded.*

Proof. Per Equation 8.2, summaries are drawn from the set

$$\{1, \ldots, 2(l+1)\} \longrightarrow Q^{2i-2} \times Q^{2i-1}$$

where l is the maximum number of children that may be born to a node at level $2i$. By definition this does not exceed B. Taking the size of Q as a constant in the automaton, the total size of the set of possible summaries is at worst exponential in B. □

Since the number of summaries is bounded, it shall suffice to enumerate and check each candidate summary in turn. Observe that enumerating summaries is simple; checking the summaries is performed as follows.

We compute $\mathit{Summary}(\mathcal{A}, 2i)$ assuming that $\mathit{Summary}(\mathcal{A}, 2i+2)$ is known. We reduce testing emptiness of $\mathcal{B}^{\uparrow}(2, \mathit{Summary}(\mathcal{A}, 2i+2))$ from Lemma 8.3 to reachability on a VASS. Since constructing a VASS directly would not make for a readable proof, we instead present a nondeterministic algorithm written in a simple procedural pseudo-langauge. The program will use variables ranging over bounded domains, and some fixed set of non-negative counters. By construction, every counter will be tested for 0 only at the end of the computation.

The simplicity of this pseudo-language is such that it can be encoded directly into the transition system of a VASS, in such a way that acceptance by the program is equivalent to reachability of a particular configuration in that VASS.

The variables of the program are as follows:

$$\begin{aligned}
\widehat{r} &\in \mathsf{dom}(\widehat{f}) && \\
\mathit{state} &\in Q_{2i} \cup \{\bot\} && \\
\mathit{state}[j] &\in Q_{2i+1} \cup \{\bot, \top\} && j \in \{1, \dots, b\} \\
\mathit{children}[j, f, r] &\in \mathbb{N} && f \text{ summary at level } (2i+2),\ r \in \mathsf{dom}(f)
\end{aligned}$$

Intuitively, *state* and $\widehat{r}$ represent a state from $Q^{(0)}$ of $\mathcal{B}^{\uparrow}(2, \mathit{Summary}(\mathcal{A}, 2i+2))$. The initial configuration is empty, so $\mathit{state} = \bot$. Each variable $\mathit{state}[j]$ represents the state of the jth child of the root. By the boundedness property, the root will never have more than b children in its lifetime, so we can give each child a unique index from $\{1, \dots, b\}$. A value of $\bot$ for $\mathit{state}[j]$ means that the j-th child has not yet been yet added, and $\mathit{state}[j] = \top$ indicates that that child has been removed. The counter $\mathit{children}[j, f, r]$ indicates the number of children of the j-th child of the root with a particular summary f of level $2i + 2$ and usage counter r. Since we are simulating LLA, it follows that $\mathit{children}[j, f, r]$ will always be zero when $\mathit{state}[j] \in \{\bot, \top\}$.

Following these intuitions the initial values of the variables are $\widehat{r} = 1$, $\mathit{state} = \bot$, $\mathit{state}[j] = \bot$ for every j, and $\mathit{children}[j, f, r] = 0$ for every j, f, and r.

The program $\mathsf{TEST}(\widehat{f})$ we are going to write is a set of rules to be executed nondeterministically. Either the program will eventually ACCEPT, or it will block with no further rules that can be applied. We later show that the program accepts for $\widehat{f}$ if and only if $\widehat{f} \in \mathit{Summary}(\mathcal{A}, 2i)$. The rules of the program are derived from the transitions of $\mathcal{A}$ and $\mathcal{B}^{\uparrow}(2, \mathit{Summary}(\mathcal{A}, 2i+2))$ from Lemma 8.3. They are formally defined as follows.

Initializing the root We have a rule

```
    if   $\mathit{state} = \bot$
    then $\mathit{state} := q'^{(2i)}$
         $\widehat{r} := 3$
```

for every transition

$$f(1) \xrightarrow{a} (f(2), q'^{(2i)}) \in \delta_{\mathsf{Q}}^{(2i)}.$$

Removing the root and accepting. The program is able to accept when it has completed all simulated interactions. Observe that this is the only time that the counters are tested for zero. Since this occurs at the end of the program, it can be easily checked by VASS reachability.

```
  if  $state = q^{(2i)}$
      $\widehat{r} = \mathsf{max}(\mathsf{dom}(\widehat{f})) - 1$
      $\forall j\colon state[j] = \top$
      $\forall (j, f, r)\colon children[j, f, r] = 0$
then  $\mathsf{ACCEPT}$
```

for every transition

$$(f(\widehat{r}), q^{(2i)}) \xrightarrow{a} f(\widehat{r}+1) \in \delta_{\mathsf{A}}^{(2i)}.$$

Adding a node at level $2i+1$. We ensure that we are in the correct state and ensure that the summary we are testing aligns with some transition from the automaton.

```
  if  $state = q^{(2i)}$
      $\widehat{f}(\widehat{r}) = (q^{(2i-2)}, q^{(2i-1)})$
      $\widehat{f}(\widehat{r}+1) = (q'^{(2i-2)}, q'^{(2i-1)})$
      $\widehat{r} + 2 < \mathsf{max}(\mathsf{dom}(\widehat{f}))$
      $\exists j\colon state[j] = \bot$
then  $state := q'^{(2i)}$
      $state[j] := q'^{(2i+1)}$
      $\widehat{r} := \widehat{r} + 2$
```

for every transition

$$(q^{(2i-2)}, q^{(2i-1)}, q^{(2i)}) \xrightarrow{t} (q'^{(2i-2)}, q'^{(2i-1)}, q'^{(2i)}, q'^{(2i+1)}) \in \delta_{\mathsf{Q}}^{(2i+1)}.$$

Removing a node at level $2i+1$. We delete a child according to some transition from $\delta_{\mathsf{Q}}^{(2i+1)}$. While the zero test (ensuring j is a leaf) is not performed here directly, no further operations will be made on children counters of this child and hence a zero test performed at the end of the simulation is equivalent to one performed at the point of removal.

```
    if   $state = q^{(2i)}$
         $\widehat{f}(\widehat{r}) = (q^{(2i-2)}, q^{(2i-1)})$
         $\widehat{f}(\widehat{r}+1) = (q'^{(2i-2)}, q'^{(2i-1)})$
         $\widehat{r} + 2 < \mathsf{max}(\mathsf{dom}(\widehat{f}))$
         $\exists j\colon state[j] = q^{(2i+1)}$
    then $state := q'^{(2i)}$
         $state[j] := \top$
         $\widehat{r} := \widehat{r} + 2$
```

for every transition

$$(q^{(2i-2)}, q^{(2i-1)}, q^{(2i)}, q^{(2i+1)}) \xrightarrow{t} (q'^{(2i-2)}, q'^{(2i-1)}, q'^{(2i)}) \in \delta_{\mathsf{A}}^{(2i+1)}.$$

Adding a node at level $2i+2$. Firstly we ensure that there is some child j where such a node can be appended. We simulate creation of a child by nondeterministically choosing a summary and increasing the corresponding unbounded counter. As described earlier in the proof, the child jumps to index 3 (see the use of *children*$[j, f, 3]$) because we automatically perform the initial pair of reads and writes described in that summary when adding the given node.

```
    if   $state = q^{(2i)}$
         $\exists j\colon state[j] = q^{(2i+1)}$
    then $state := q'^{(2i)}$
         $state[j] := q'^{(2i+1)}$
         $children[j, f, 3] \mathrel{+}= 1$
         for some $f \in Summary(2i+2)$
           s.t. $f(1) = (q^{(2i)}, q^{(2i+1)})$
           and $f(2) = (q'^{(2i)}, q'^{(2i+1)})$
```

Progressing a child at level $2i+2$. We identify an appropriate child j which itself has a child in state (f, r). We use the test $r + 2 < \mathsf{max}(\mathsf{dom}(f))$ to ensure that the last interaction of the node is reserved for deletion of our root node.

```
    if   $state = q^{(2i)}$
         $\exists(j,f,r)\colon$ $state[j] = q^{(2i+1)}$
                  and $f(r) = (q^{(2i)}, q^{(2i+1)})$
                  and $f(r+1) = (q'^{(2i)}, q'^{(2i+1)})$
                  and $(r+2) < \mathsf{max}(\mathsf{dom}(f))$
                  and $children[j,f,r] \geq 1$
  then   $state := q'^{(2i)}$
         $state[j] := q'^{(2i+1)}$
         $children[j,f,r+2]$ += 1
         $children[j,f,r]$ −= 1
```

Observe that the test $children[j,f,r] \geq 1$ can be simulated by a VASS because we have $children[j,f,r]$ −= 1 as a consequence.

Removing a node at level $2i+2$. We find a child which has completed its summary to the point that it can now be removed. We use the last values in f to determine how to remove the node.

```
    if   $state = q^{(2i)}$
         $\exists(j,f,r)\colon$ $state[j] = q^{(2i+1)}$
                  and $f(r) = (q^{(2i)}, q^{(2i+1)})$
                  and $f(r+1) = (q'^{(2i)}, q'^{(2i+1)})$
                  and $(r+1) = \mathsf{max}(\mathsf{dom}(f))$
                  and $children[j,f,r] \geq 1$
  then   $state := q'^{(2i)}$
         $state[b] := q'^{(2i+1)}$
         $children[j,f,r]$ −= 1
```

Lemma 8.5. *Program* $\mathsf{TEST}(\widehat{f})$ *accepts if and only if* $\widehat{f} \in Summary(\mathcal{A}, 2i)$.

Proof. By definition, $\widehat{f} \in Summary(\mathcal{A}, 2i)$ if automaton $\mathcal{B} = \mathcal{A}^{\downarrow}(2i, \widehat{f})$ accepts a trace. By Lemma 8.3 this is equivalent to $\mathcal{B}^{\uparrow}(2, Summary(\mathcal{A}, 2i+2))$ accepting some trace. It can be checked that the instructions of $\mathsf{TEST}(\widehat{f})$ correspond one-to-one to transitions of $\mathcal{B}^{\uparrow}(2, Summary(\mathcal{A}, 2i+2))$. So an accepting run of $\mathsf{TEST}(\widehat{f})$ can be obtained from a trace accepted by $\mathcal{B}^{\uparrow}(2, Summary(\mathcal{A}, 2i+2))$, and vice versa. □

Completing the Induction

Since we now have a terminating process for determining the summaries at level $2i$ based on summaries at level $2i+2$, we can proceed by induction to reduce from emptiness of a k-LLA to emptiness of the corresponding 3-LLA which summarises it.

The base case for this induction is the original k-LLA. We assumed (without loss of generality) that k is even, and hence the lowest level of the automaton $(k-1)$ is an odd level. The summaries at the level above, the bottom-most even level of the automaton, can be computed easily as follows.

Since nodes in level $(k-1)$ of the automaton will never have children, their state can never change, per the definition of LLA. The level above that, which is even, can then follow the same procedure for the computation of summaries as described in **Computing summaries**, with the simplification that $\mathit{children}[j, f, r]$ is always zero and all sections related to "nodes at level $2i+2$" can be ignored. (Indeed, since this elides all unbounded counters, the resulting automaton is completely finite and checking reachability is trivial.)

Emptiness of the 3-LLA

The final 3-LLA consists of levels 0 and 1, exactly as they appear in $\mathcal{A}$, with levels 2 and below being summarized at level 2. Previously we have worked with the assumption that grandparents exist for any given node; at low levels this is not true. In particular, we are unable to construct summaries at level 0, since summaries (as described earlier in this proof) relate to the levels above, and there are no levels above 0.

Instead, for this final case, we modify the procedure described in **Computing summaries** such that initialising the root, adding a node at level $2i+1$ (i.e. 1), and removing a node at level $2i+1$ are not preconditioned on any $\widehat{f}$. The simulation of level 2 proceeds as normal, since that level does not refer above level 0 ($\widehat{f}$ does not appear).

This minor change is sufficient to use the **Computing summaries** procedure to conclude the emptiness problem: the k-LLA $\mathcal{A}$ is nonempty if and only if the summary procedure on our final summarising 3-LLA returns a nonempty set of summaries for level 0. Since we apply the summarisation procedure a bounded number of times, each of which terminates, this procedure as a whole will terminate. $\square$

8.3 Local FICA

With the “local” of local leafy automata now fully defined, we can proceed to the fragment of FICA which corresponds to this restriction.

The identification of such a fragment is propelled by the translation from FICA to LA defined in Chapter 7. We will reiterate the essential structure of that translation now.

Depth in the automaton is introduced by function application in the FICA term in question. More precisely, each function application (call it $f(M)$) introduces two new levels below its corresponding node (call it d at level $2i$).

- Level $2i + 1$ is bounded in the number of nodes spawned as a child of d across a run. The nodes spawned and their transitions are uncomplicated and defined exactly by the translation from FICA to LA.
- The level below that ($2i + 2$) is unbounded in the number of nodes spawned as a child of those child nodes at level $2i + 1$. The level-$2i + 2$ nodes correspond to the unbounded number of threads spawned which may evaluate M.

This is the only source of depth in the automaton, with the exception of transient read/write actions which may be replaced by ε-transitions (and in any case whose added depth does not exceed 1).

Recall also from the Chapter 7 translation that a **newvar** ... **in** M or **newsem** ... **in** M expression induced a product of states at the root of the subtree corresponding to M. Hence the notion of variable scope is already encoded in the original LA translation.

We shall now give two equivalent definitions of the restrictions over FICA. The first (below) is less formal.

Definition 8.4. A FICA term $\Gamma \vdash M : \theta$ is in LFICA if:

1. M does not contain **while**; and
2. there is no variable x, free variables f, g, and binder **new** $\in \{$**newvar**, **newsem**$\}$ such that any subterm of the form

$$\mathbf{new}\ x := i\ \mathbf{in}\ \ldots f(\ldots g(\ldots x \ldots) \ldots) \ldots$$

appears in M.

Constraint 1 gives us boundedness at even levels. Currently **while** is the only construct which allows unbounded iteration in a term, and hence the only construct by which unboundedly many children may be spawned by an even-level node in a leafy automaton. In removing **while**, we bound all even levels immediately.

Intuitively, constraint 2 prevents the scope of a variable x from traversing two occurrences of free variables. Recalling the translation from Chapter 7, this corresponds to information within the leafy automaton not being able to traverse two even layers in the same transition. It should now be clear how that restriction maps to the structural constraint on `LLA`.

Per the two restrictions given above, the following terms are in `LFICA`:

(a) $\mathbf{newvar}\, x := 0\, \mathbf{in}\, x := 1 \,||\, f(x := 2)$,

(b) $\mathbf{newvar}\, x := 0\, \mathbf{in}\, f(\mathbf{newvar}\, y := 0\, \mathbf{in}\, f(y := 1) \,||\, x := !y)$

But the following terms are not in `LFICA`:

(c) $\mathbf{newvar}\, x := 0\, \mathbf{in}\, \mathbf{while}\, !x = 0\, \mathbf{do}\, x := succ(!x)$

(d) $\mathbf{newvar}\, x := 0\, \mathbf{in}\, f(f(x := 1))$

as they violate conditions 1 and 2 respectively.

We shall now redefine `LFICA` formally by induction over `FICA` terms. Let us introduce the notion of *applicative depth* of a variable $x : \beta$ inside some term Q in $\beta\eta$-normal form. This is a measure of the number of free variables through occurrences of which x is visible. The applicative depth of x is defined inductively as given by Table 8.1.

shape of Q	$ad_x(Q)$
x	1
$y\ (y \neq x)$, **skip**, **div**, i	0
$\mathbf{op}(M)$, $!M$, $\mathbf{release}(M)$, $\mathbf{grab}(M)$	$ad_x(M)$
$M; N$, $M \vert\vert N$, $M := N$, $\mathbf{while}\, M\, \mathbf{do}\, N$	$\mathsf{max}(ad_x(M), ad_x(N))$
$\mathbf{if}\, M\, \mathbf{then}\, N_1\, \mathbf{else}\, N_2$	$\mathsf{max}(ad_x(M), ad_x(N_1), ad_x(N_2))$
$\lambda y.M$, $\mathbf{new}\, y := i\, \mathbf{in}\, M$	$ad_x(M[z/y])$, where z is fresh
$f M_1 \dots M_k$	$1 + \mathsf{max}(ad_x(M_1), \dots, ad_x(M_k))$

Table 8.1: Inductive definition of $ad_x(Q)$, the applicative depth of variable x in term Q. Let $\mathbf{new} \in \{\mathbf{newvar}, \mathbf{newsem}\}$.

Per Table 8.1, terms (a)-(d) have applicative depth of x 2, 2, 1, and 3, respectively[3]. In (b), the applicative depth of y is also 2.

Definition 8.5. A FICA term $\Gamma \vdash M : \theta$ is in LFICA if:

- **while** does not appear in M; and
- for every subterm of shape $\mathbf{newvar}\, x := i \,\mathbf{in}\, N$ or $\mathbf{newsem}\, x := i \,\mathbf{in}\, N$ in the $\beta\eta$-normal form of M, we have $ad_x(N) \leq 2$.

It should be clear that the formal definition (8.5) and the natural definition (8.4) are equivalent.

8.3.1 From FICA to LA

We now proceed to show that this notion of local FICA terms is compatible with the restrictions of LLA.

Theorem 8.2. *For any LFICA term $\Gamma \vdash M : \theta$, the* LA $\mathcal{A}_M$ *obtained by the translation given in the proof of Theorem 7.2 can be presented as an* LLA $\mathcal{A}'_M$*. Moreover, the recovered* LLA *is even-bounded.*

Proof. The proof makes significant use of the locality condition (condition 2) of LFICA and the fact that nontrivial depth in the automaton is introduced only by function application. We again give an inductive argument, in this case showing that the translation for each term preserves the locality conditions of LLA and the even-boundedness property.

We shall introduce some shorthand notation to simplify the inductive translation. We will describe an LA as being LLAcompatible if its transitions are conducive to translation to LLA.

Let $j^{\uparrow}$ be $j-2$ for even j and $j-3$ for odd j.

Definition 8.6. A set of transitions $C^{(j)} \subseteq \delta^{(j)}$ is LLA-*compatible* iff

- Every transition in $C^{(j)}$ is of the shape

$$(q^{(0)}, \cdots, q^{(j^{\uparrow}-1)}, q^{(j^{\uparrow})}, \cdots, q^{(j)}) \xrightarrow{\mathsf{m}} (q^{(0)}, \cdots, q^{(j^{\uparrow}-1)}, r^{(j^{\uparrow})}, \cdots, r^{(j')}) \tag{8.3}$$

[3]This is assuming *succ* to be some known function and not a free variable.

- For every transition in the shape of (8.3), and every $(s^{(0)}, \cdots, s^{(j^{\uparrow}-1)}) \in Q^{(0)} \times \cdots \times Q^{(j^{\uparrow}-1)}$, the transition

  $$(s^{(0)}, \cdots, s^{(j^{\uparrow}-1)}, q^{(j^{\uparrow})}, \cdots, q^{(j)}) \xrightarrow{\mathsf{m}} (s^{(0)}, \cdots, s^{(j^{\uparrow}-1)}, r^{(j^{\uparrow})}, \cdots, r^{(j')})$$

  must also appear in $C^{(j)}$.

If, for every i, $\delta_M^{(i)}$ can be partitioned into LLA-compatible subsets, then $\mathcal{A}_M$ is LLA-compatible.

Lemma 8.6. *Any* LLA*-compatible* LA*,* $\mathcal{A}$*, can be translated (in log-space) to an* LLA*,* $\mathcal{A}'$*, such that* $\mathcal{A}$ *and* $\mathcal{A}'$ *have the same set of accepting traces. If* $\mathcal{A}$ *is even-bounded then* $\mathcal{A}'$ *is also even-bounded.*

Proof. If an LA is LLA-compatible, then all its transitions are such that, for any j, the states above $j^{\uparrow}$ convey no information about whether any transition in $\delta^{(j)}$ can be fired, and those states are not affected by any such transition. That being the case, the following translation rule can be used to convert any LA transition in $\delta^{(j)} (j > 1)$ to an LLA transition in $\delta'^{(j)}$:

$$\frac{(q^{(0)}, \cdots, q_1^{(j^{\uparrow}-1)}, q_1^{(j^{\uparrow})}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (q^{(0)}, \cdots, q_1^{(j^{\uparrow}-1)}, r_1^{(j^{\uparrow})}, \cdots, r_1^{(j')})}{(q_1^{(j^{\uparrow})}, \cdots, q_1^{(j)}) \xrightarrow{\mathsf{m}} (r_1^{(j^{\uparrow})}, \cdots, r_1^{(j')})}$$

The translation of the initial and final transitions is immediate:

$$\frac{\dagger \xrightarrow{q} r^{(0)}}{\dagger \xrightarrow{q} r^{(0)}} \qquad \frac{q^{(0)} \xrightarrow{a} \dagger}{q^{(0)} \xrightarrow{a} \dagger}$$

□

It remains to show that LLA-compatibility and even-boundedness is maintained by each construction from the proof. For the sake of symmetry, the cases in this proof are given in the same order and style as the cases in the proof of Theorem 7.2. The cases of particular import to this proof are $\Gamma \vdash f M_h \cdots M_1 : \beta$ (function application) and $\Gamma \vdash \mathbf{newvar}\, x := i\, \mathbf{in}\, M : \beta$ (variable binding).

$$\Gamma \vdash \mathbf{skip} : \mathbf{com} \qquad \Gamma \vdash \mathbf{div_{com}} : \mathbf{com} \qquad \Gamma \vdash \mathbf{div}_{\theta} : \theta \qquad \Gamma \vdash i : \mathbf{exp}$$

These base cases do not interact with the locality condition and their translation rules are LLA-compatible. They are trivially even-bounded.

$$\Gamma \vdash \mathbf{op}(M_1) : \mathbf{exp}$$

The only modification made on seeing this term is to label the i-transition marking the returned value by $\widehat{\mathbf{op}}(i)$ instead. This does not interact with the locality condition and hence the **op**-adjusted transitions are LLA-compatible. As no new transitions are introduced, they are even-bounded by the inductive hypothesis.

$$\Gamma \vdash M_1 || M_2 : \mathbf{com}$$

Recall from the proof that running M_1 and M_2 in parallel requires us to take the product of the states at level 0 and the disjoint sum of levels $>$ 0. Moreover the states at level 0 corresponding to each of M_1 and M_2 do not interact with each other; and this translation does not itself induce any extra levels or any unboundedness into the automaton. Hence the locality condition is preserved by parallel composition and induced transitions are LLA-compatible. The number of children spawned at level 1 will be the sum of the children spawned by M_1 and M_2 individually, and hence is still even-bounded.

$$\Gamma \vdash M_1; M_2 : \beta$$

As with parallel composition, sequential composition does not introduce any new depth to the machine and so the transitions are LLA-compatible by the inductive hypothesis. Again as per number of children spawned at level 1 will be the sum of the children spawned by M_1 and M_2 individually, and hence is still even-bounded.

$$\Gamma \vdash \mathbf{newvar}\, x := i \,\mathbf{in}\, M : \beta$$

By the inductive hypothesis, $\Gamma \vdash M : \beta$ is an $\mathtt{LFICA}$ term, and therefore so is $\Gamma, x : \mathbf{var} \vdash M : \beta$, and $ad_x(M) \leq 2$.

Occurrences of x at applicative depth 1 will induce reads and writes (i.e. $\mathsf{read}^{(x,\rho)}$, $\mathsf{write}(i)^{(x,\rho)}$, $\mathsf{ok}^{(x,\rho)}$ and $i^{(x,\rho)}$ transitions) which add and remove nodes at level 1. Occurrences of x at applicative depth 2 will induce reads and writes which add and remove nodes at level 3. The number of reads and writes (hence the number of nodes created in this way) is bounded by the number of occurrences in the term itself, since **while** is forbidden.

In the $\mathbf{newvar}\, x := i \,\mathbf{in}\, M$ construction, the state is stored at the root by a product construction, and reads and writes are simulated by modifying the component of the root's state corresponding to that variable. Since $j^{\uparrow}$ is 0 for $j = 3$, the reads and writes at levels 1 and 3 are allowed to interact with the state at the root. So this construction preserves locality and even-boundedness without interfering with the simulation of variables. All transitions unrelated to x are not affected by this construction and trivially maintain both even-boundedness and LLA-compatibility.

The logic for $\Gamma \vdash \mathbf{newsem}\, x := i \,\mathbf{in}\, M : \beta$ is symmetrical.

$$\Gamma \vdash f M_h \cdots M_1 : \beta$$

The function application construction adds two new levels at the "top" of the automaton. The topmost level defines control flow through the machine and admits exactly one child at level 1, which is then used to spawn an unbounded number of copies of the automata for M_u ($u \in \{1, \ldots, h\}$). Since the automata for M_u are shifted down by exactly 2 levels, and (by the inductive hypothesis) their LA are already even-bounded, even-boundedness is preserved.

There is a minor wrinkle in the LLA-compatibility condition of the original translation from the proof. The translation is reproduced here:

$$\frac{(q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(\vec{i},\rho)}}_u (q_u^{(0)}, \cdots, q_u^{(j')})}{(1, 0, q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(fu\vec{i},\rho)}} (1, 0, q_u^{(0)}, \cdots, q_u^{(j')})}$$

Observe that this translation specifies (1,0) as the states at level 0 and 1. However, it is a structural property of this particular translation that children at level 2 and below can only ever exist when the level-0 state is 1; and $Q^{(1)} = \{0\}$. So we can rewrite this rule as

$$\frac{(q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(\vec{i},\rho)}}_u (q_u^{(0)}, \cdots, q_u^{(j')}) \qquad \phi \in \{0, 1, 2\}}{(\phi, 0, q_u^{(0)}, \cdots, q_u^{(j)}) \xrightarrow{m^{(fu\vec{i},\rho)}} (\phi, 0, q_u^{(0)}, \cdots, q_u^{(j')})}$$

which produces an LLA-compatible set of transitions.

$$\Gamma \vdash \textbf{if}\, M_1\, \textbf{then}\, M_2\, \textbf{else}\, M_3 : \beta$$

As with sequential composition, even-boundedness is maintained—the new bound is the bound for M_1 plus the greater of the bounds of M_2 and M_3. Since the construction simulates M_1 followed by one of M_2 and M_3, LLA-compatibility follows by the inductive hypothesis.

$$\Gamma \vdash \textbf{while}\, M_1\, \textbf{do}\, M_2 : \textbf{com}$$

This construct is forbidden in LFICA; we shall take a second to show why.

Consider this term:

$$\textbf{newvar}\, x := 1\, \textbf{in}\, \textbf{while}\, x = 0\, \textbf{do}\, x := succ(!x)$$

In this term, the *while* loop will never terminate, and the *do* block creates at least one child while reading and writing x. Hence we cannot place an upper bound on the number of children spawned by this term, and since this term would exist at an even level, it would violate the even-boundedness property, which is required for decidability of emptiness in LLA.

$$\Gamma \vdash !M_1 : \textbf{exp}$$

Dereferencing relabels some transitions but otherwise has no impact on the running of the automaton; even-boundedness and LLA-compatibility follow trivially.

$$\Gamma \vdash M_1 := M_2 : \textbf{com}$$

As with sequential composition, the new even bound is the sum of the bounds of M_1 and M_2, and LLA-compatibility is not impacted by the translation.

$$\Gamma \vdash \textbf{grab}(M_1) : \textbf{com} \text{ and } \Gamma \vdash \textbf{release}(M_1) : \textbf{com}$$

These cases induce a renaming on some transitions but otherwise do not impact the automaton. Even-boundedness and LLA-compatibility follow. □

Corollary 8.1. *For any* LFICA *term* $\Gamma \vdash M : \theta$*, the problem of determining whether* comp($[\![\Gamma \vdash M]\!]$) *is empty is decidable.*

The corollary is implied by Theorems 7.1, 8.1, and 8.2. Moreover, Theorem 7.1 gives us that checking for emptiness is equivalent to checking equivalence with a term which always diverges. In cases where this equivalence does not hold, we can (by the argument for Theorem 8.1) extract some distinguishing trace, which can then be fed to the Definability Theorem (Theorem 41, [70]). This is a valuable property, since in the concurrent setting bugs can be difficult to replicate. Such a mechanism allows us to explore the context in which a property is violated, hence (in a verification setting) not only revealing to the end user that a bug exists but also the nature of the setting in which it may occur or be exploited.

Chapter 9

Split Automata

Restricting communication in leafy automata led to some interesting and meaningful decidability results in Chapter 8. In the following work we continue in the same spirit, identifying a new way to augment the tree-based automaton structure with new information and at the same time limiting branch-based communication. In doing so, we shall find a new way to represent certain FICA terms and effective procedures for deciding verification problems over such terms.

Like leafy automata and LLA, split automata are defined over an infinite data forest $\mathcal{D}$, which maintains the infinite branching and well-foundedness properties described in Section 6.2. To fully enjoy this chapter, it is recommended to (re-)familiarize oneself with the preliminaries of game semantics found in Section 7.2.

9.1 Control versus Memory

Configurations in split automata are slightly different to those in LA and LLA. A configuration will still be defined as a finite subtree of $\mathcal{D}$ labelled with states. However, in this case, states consist of exactly one control state, and zero or more *memory cells*. Memory is kept only at even levels; nodes at odd levels will never have memory cells associated to them.

Control states and memory cells are interacted with differently and separately by transitions in the machine. Control states are reminiscent of the states attached to nodes in LA/LLA, with the exception that only the immediate parent may be referred to when adding or removing a node. Transitions which add or remove nodes have no access to the memory cells at all.

Memory cells are interacted with entirely through a new form of transition called EPS (short for "epsilon"). Transitions of this form allow for the read and/or write of any one control state at some node n, and at the same time to read and/or write some memory cell on the path from the root to n. As with ϵ-transitions in other automata models, EPS transitions do not read the input, so an arbitrary number of such transitions may be fired. EPS transitions have no access to any other control states.

In this way, we cleanly divide the logic of the program between transitions which read the input, and always add or remove a leaf with only local access; and transitions which do not read the tape but have more power. As we shall see, this structural separation has meaningful implications for our representation of FICA terms.

Each split automaton is parametrized by two values, k and N. The parameter k is the depth of the automaton, as in our previous models—we may occasionally refer to k-SA in the same fashion, meaning an SA with depth parameter k. N is the maximum number of memory cells stored at each state. For the sake of simplicity, we shall fix that each node has the same number of memory cells; not all such memory cells will be used.

9.2 Split Automata

The complete formal definition of split automata is as follows.

Definition 9.1. A *split automaton* (SA) is a tuple $\mathcal{A} = \langle \Sigma, k, N, C, \delta \rangle$, where:

- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is a finite alphabet, partitioned into questions and answers;
- $k \geq 0$ is the depth parameter;
- $N \geq 0$ is the local memory capacity;
- $C = \bigcup\{C^{(i)} : i = 0, \ldots, k\}$ is a finite set of *control states*, partitioned into sets $C^{(i)}$ of level-i control states;
- δ is the transition function, with transitions in δ partitioned according to their type and level on which they operate. (In what follows, take $c^{(i)}, d^{(i)} \in C^{(i)}$):
  - ADD(i) transitions are $c^{(i-1)} \xrightarrow{q} (d^{(i-1)}, d^{(i)})$, and $\dagger \xrightarrow{q} c^{(0)}$ for the special case of $i = 0$, with $q \in \Sigma_{\mathsf{Q}}$,
  - DEL(i) transitions are $(c^{(i-1)}, c^{(i)}) \xrightarrow{a} d^{(i-1)}$, and $c^{(0)} \xrightarrow{a} \dagger$ for the special case of $i = 0$, with $a \in \Sigma_{\mathsf{A}}$,
  - EPS($2j, 2i$) transitions read $v \in V$ from memory cell $h \in \{1, \ldots, N\}$ at level $2j \leq 2i$ and update it to $v' \in V$, but do not read the input: $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$.

Transitions cannot modify control states at odd layers: if
$c^{(2i-1)} \xrightarrow{q} (d^{(2i-1)}, d^{(2i)}) \in \delta$ or $(c^{(2i-1)}, c^{(2i)}) \xrightarrow{a} d^{(2i-1)} \in \delta$ then
$c^{(2i-1)} = d^{(2i-1)}$.

9.2.1 Configurations and Transitions

A configuration of a split automaton is a tuple (D, E, f, m), where:

- D is a finite subset of $\mathcal{D}$, consisting of those data values that have been read so far in a word;
- E is a finite subtree of $\mathcal{D}$ rooted at a level-0 data value, representing the shape of the tree in the current configuration;
- $f : E \to C$ is a level-preserving function[1] which maps data values to control states; and
- $m : E \rightharpoonup V^N$ is a partial function which maps each even-level data value onto a vector representing the current values of its node's memory cells.

A split automaton starts reading a word in the empty configuration $\kappa_0 = (\emptyset, \emptyset, \emptyset, \emptyset)$ and proceeds according to the transitions of δ as explained below. Let $\kappa = (D, E, f, m)$ be the current configuration.

With ADD and DEL transitions, the main difference between the behaviour of split automata and leafy automata (local or otherwise) is that only the parent node may be read or written when adding or removing a leaf. In other respects they function the same as question and answer transitions (respectively) from the former models.

We shall write $f[\cdots]$ for an extension or update of f. More precisely, $f[x \mapsto y]$ is exactly $(f \setminus \{(x, f(x))\}) \cup \{(x, y)\}$. The same notation is extended to arbitrary numbers of updates, so $f[x \mapsto y, a \mapsto b]$ means the same as $f[x \mapsto y][a \mapsto b]$. We shall also use $f[x \mapsto y]$ as a precondition on a transition; those cases are discussed separately later in the chapter.

An ADD transition from κ is possible on a data letter (t, d) when $t = q \in \Sigma_{\mathsf{Q}}$ is a question letter and $d \notin D$ is a fresh level-i data value such that $\mathit{pred}(d) \in E$ (the parent of d is in the configuration). In this case, the automaton adds a new leaf d to the configuration and updates the control state. The new leaf gets the control state determined by the transition. If it is on an even level, its memory is initialised. Formally, $\mathcal{A}$ goes from κ to $\kappa' = (D \cup \{d\}, E \cup \{d\}, f', m')$ provided one of the following conditions holds:

[1]By *level-preserving* we mean that if D is a level-i data value then $f(d) \in C^{(i)}$.

- On transition $c^{(i-1)} \xrightarrow{q} (d^{(i-1)}, d^{(i)})$ when $f(\mathit{pred}(d)) = c^{(i-1)}$, $f' = f[\mathit{pred}(d) \mapsto d^{(i-1)}, d \mapsto d^{(i)}]$, and

$$m' = \begin{cases} m & \text{if } i \text{ is odd} \\ m[d \mapsto 0^N] & \text{if } i \text{ is even} \end{cases}$$

- On transition $\dagger \xrightarrow{q} d^{(0)}$ when $D = \emptyset$, $f' = [d \mapsto d^{(0)}]$, and $m' = [d \mapsto 0^N]$.

On reading a data letter (t, d) with $t = a \in \Sigma_{\mathsf{A}}$ an answer letter and $d \in E$ a level-i data value, a transition is possible only if d is a leaf in E. A DEL transition deletes d and updates the parent control state without modifying the associated memory (if any). Formally, the automaton changes its configuration to $\kappa' = (D, E \setminus \{d\}, f' \setminus \{d\}, m \setminus \{d\})$ provided one of the following conditions holds:

- On transition $(c^{(i-1)}, c^{(i)}) \xrightarrow{a} d^{(i)}$ when $f(\mathit{pred}(d)) = c^{(i-1)}$, $f(d) = c^{(i)}$, and $f' = f[\mathit{pred}(d) \mapsto c^{(i-1)}]$.
- On transition $c^{(0)} \xrightarrow{a} \dagger$ when $f' = f$.

Observe that the last transition is possible only when d is a leaf of E and at level 0 at the same time; the result of the transition is that E is the empty tree.

EPS transitions are silent—they do not read a data letter from the input. They apply at even levels only and do not change the data values seen (D) nor the tree (E). However, they may read/write one control state and read/write one memory location situated at the same level or another even level above. Formally, the automaton can go to $\kappa' = (D, E, f', m')$ on transition $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$ if there is a level-$2i$ data value $d \in E$ such that $m(\mathit{pred}^{2i-2j}(d))(h) = v$, $f(d) = c^{(2i)}$, $f' = f[d \mapsto d^{(2i)}]$ and $m'(\mathit{pred}^{2i-2j}(d)))(h) = v'$, and m' is the same as m otherwise.

Note that the $2j$ parameter is used as an *offset* from the level of the control state; if $2j = 0$ then we modify the memory cells and the control state at the same node.

We use analogous definitions of language, traces and accepting traces to those in Chapters 6 and 8. In particular, a configuration is accepting if the data subtree E is empty, and a trace is accepting if some sequence of configurations exists with transitions corresponding to the data letters of the trace and with the final configuration accepting. The language $L(\mathcal{A})$ of an automaton $\mathcal{A}$ is the set of words resulting in accepting traces. Likewise, game-semantic notions and terminology are shared between earlier chapters and this work.

9.3 From FICA to SA

The modified structure of split automata, as compared to (local) leafy automata, is motivated by finding a good fit for the game semantics of FICA, and in particular finding a model which neatly accommodates a similar translation to the one outlined in the previous chapter. Like LA, but unlike LLA, SA are able to express the full game semantics of FICA. As such most verification problems (including the emptiness problem) are undecidable.

We shall now provide the translation from FICA terms to split automata in full. Afterwards, we shall evaluate this translation in comparison to the original translation to LA. Some (but not all) of the notation and high-level concepts used will be analogous.

We begin by recalling the game-semantic notions that underpin the translation. Within the game semantics of FICA the moves, arenas, and terms-in-context are exactly as described at the beginning of Section 7.3 and we shall use those definitions from hereon out.

Example 9.1. Consider the split automaton $\mathcal{A} = \langle Q, 3, 0, \Sigma, \delta\rangle$, where

- $Q^{(0)} = \{0,1,2\}$, $Q^{(1)} = \{3\}$, $Q^{(2)} = \{4,5,6\}$, $Q^{(3)} = \{7\}$,
- $\Sigma_{\mathsf{Q}} = \{\mathsf{run}^{(\epsilon,0)}, \mathsf{run}^{(f,0)}, \mathsf{run}^{(f1,0)}, \mathsf{run}^{(x,2)}\}$,
- $\Sigma_{\mathsf{A}} = \{\mathsf{done}^{(\epsilon,0)}, \mathsf{done}^{(f,0)}, \mathsf{done}^{(f1,0)}, \mathsf{done}^{(x,0)}\}$, and
- δ is given by

$$\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}} 0 \qquad 0 \xrightarrow{\mathsf{run}^{(f,0)}} (1,3) \qquad (1,3) \xrightarrow{\mathsf{done}^{(f,0)}} 2$$
$$2 \xrightarrow{\mathsf{done}^{(\epsilon,0)}} \dagger \qquad 3 \xrightarrow{\mathsf{run}^{(f1,0)}} (3,4) \qquad 4 \xrightarrow{\mathsf{run}^{(x,2)}} (5,7)$$
$$(5,7) \xrightarrow{\mathsf{done}^{(x,0)}} 6 \qquad (3,6) \xrightarrow{\mathsf{done}^{(f1,0)}} 3.$$

Then the traces of $Tr(\mathcal{A})$ represent exactly the plays from $\sigma = [\![f : \mathbf{com} \to \mathbf{com}, x : \mathbf{com} \vdash fx : \mathbf{com}]\!]$, including the play from Example 7.6, and $L(\mathcal{A})$ represents $\mathsf{comp}(\sigma)$.

Recalling our notion of justification pointers and the ρ based scheme from Section 7.3, one may wonder why we did not use the parent structure of $\mathcal{D}$ to represent justification pointers (this would correspond to $\rho = 0$ in all cases). Unfortunately, this simplified scheme would not work with split automata: in the above example, the number of $\mathsf{run}^{(x,2)}$ moves has to be the same as the number of $\mathsf{run}^{(f1,0)}$ moves. If we used level-1 data values for run^{x}, we would not be able to use the automaton to enforce this property.

Below we state the main result linking FICA with split automata. As per an earlier remark in this chapter, we note that question moves correspond to ADD transitions and answer moves correspond to DEL transitions. ADD transitions at odd levels are used for P-questions and at even levels O-questions; DEL transitions at odd levels are used for O-answers and at even levels P-answers. EPS transitions are neither questions nor answers.

Theorem 9.1. *For any* FICA *term* $\Gamma \vdash M : \theta$*, there exists a split automaton* $\mathcal{A}_M$ *over a finite subset of* $\mathcal{T}_{\Gamma \vdash \theta}$ *such that the set of plays represented by data words from* $Tr(\mathcal{A}_M)$ *is exactly* $[\![\Gamma \vdash M : \theta]\!]$*. Moreover,* $L(\mathcal{A}_M)$ *represents* $\mathsf{comp}([\![\Gamma \vdash M : \theta]\!])$*.*

Proof. The proof will proceed by the inductive construction over terms. The language of the constructed automata correspond to the (complete) plays of such terms. Continuing the interleaving based approach of earlier chapters, the automata allow for interleaved progression of subterms as in the game semantics, with synchronisation used to model sequential composition and related syntactic constructs like **if**.

It is well established [11] that any FICA term can be reduced to β-normal, η-long form (that is, a term that is both β-normal (there are no remaining expressions that can be reduced by applying standard function application rewrite rules), and η-long (no functions are partially applied))[2]. This proof is by induction over terms that have been normalised in this way.

In order to establish correctness we rely on two additional technical invariants to strengthen the inductive hypothesis.

P1 If the automaton reaches a configuration in which the control state at the root is c and there exists a transition $c \xrightarrow{a} \dagger$ then the root has no children, i.e. the transition can fire.

P2 Let $x : \beta$, where $\beta = \mathbf{var}, \mathbf{sem}$. The following pairs (q^x, a^x) of tags will be referred to as matching pairs: $(\mathsf{write(i)}^{(x,\rho)}, \mathsf{ok}^{(x,0)})$ and $(\mathsf{read}^{(x,\rho)}\ \mathsf{i}^{(x,0)})$ (for $\beta = \mathbf{var}$) and $(\mathsf{grb}^{(x,\rho)}, \mathsf{ok}^{(x,0)})$ or $(\mathsf{rls}^{(x,\rho)}, \mathsf{ok}^{(x,0)})$ (for $\beta = \mathbf{sem}$).

For every matching pair (q^x, a^x), if the automaton generates a trace of the form $l_1 \cdots l_h (q^x, d) l_{h+1} \cdots l_{h'} (a^x, d) l_{h'+1} \cdots l_{h''}$, where $l_j \in (\Sigma \times \mathcal{D}) + \{\epsilon\}$, then it also generates $l_1 \cdots l_h (q^x, d)(a^x, d) l_{h+1} \cdots l_{h'} l_{h'+1} \cdots l_{h''}$. Note that above we take ϵ-steps into account.

P2 is a weak version of the game-semantic saturation condition stated at the level of automata for moves corresponding to free variables of type **var** and **sem**.

[2]Reduction to β-long η-normal form may in general induce an exponential blowup; dealing with that is beyond the scope of this work.

P1 and P2 will be helpful when it comes to interpreting sequential composition and variable/semaphore binding respectively.

We use the same notation for translation rules as previously. When referring to the inductive hypothesis for a subterm M_i, we use the subscript i to refer to the automata components, e.g. $Q_i^{(j)}$, $\xrightarrow{m}_i$, etc. In contrast, $Q^{(j)}$, $\xrightarrow{m}$ will refer to the automaton that is being constructed. Inference lines indicate that wherever the premises (above the line) are fulfilled in the given components, the consequents (below the line) should be added to the automaton under construction.

We shall now proceed by induction over the structure of β-normal η-long forms. We consider each case in turn.

$\mathbf{M} \equiv \mathbf{skip.}$ $\quad k = 0$, $N = 0$, $C^{(0)} = \{0\}$.

$$\dagger \xrightarrow{\mathsf{run}} 0 \qquad 0 \xrightarrow{\mathsf{done}} \dagger$$

$\mathbf{M} \equiv \mathbf{i.}$ $\quad k = 0$, $N = 0$, $C^{(0)} = \{0\}$.

$$\dagger \xrightarrow{\mathsf{q}} 0 \qquad 0 \xrightarrow{\mathsf{i}} \dagger$$

$\mathbf{M} \equiv \mathbf{div}_\theta$**, where** $\theta \equiv \theta_1 \to \cdots \to \theta_1 \to \beta$**.** Recall that $I_{[\![\beta]\!]}$ stands for the set of initial questions in $[\![\beta]\!]$, e.g. $I_{[\![\mathbf{com}]\!]} = \{\mathsf{run}\}$. Since **div** diverges, there is no closing move. $k = 0$, $N = 0$, $C^{(0)} = \{0\}$.

$$\frac{x \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{x} 0}$$

P1 and P2 clearly hold in the above three cases, which together form the base cases of the induction.

$\mathbf{M} \equiv \mathbf{op}(\mathbf{M_1})$**.** Here we only need to adjust the final answers, i.e. $k = k_1$, $N = N_1$, $C^{(j)} = C_1^{(j)}$ $(0 \leq j \leq k)$. Formally, we copy all transitions for M_1 except DEL(0), and add the following ones.

$$\frac{c^{(0)} \xrightarrow{\mathsf{i}}_1 \dagger \quad i \in \{0, \ldots, \max\}}{c^{(0)} \xrightarrow{\widehat{\mathbf{op}}(i)} \dagger}$$

P1 and P2 are inherited.

$\mathbf{M} \equiv \mathbf{M_1} || \mathbf{M_2}$**.**

$$k = \mathsf{max}(k_1, k_2),\ N = N_1 + N_2,$$

$$C^{(0)} = C_1^{(0)} \times C_2^{(0)},\ C^{(i)} = C_1^{(i)} + C_2^{(i)}\ (i > 0).$$

This construction requires us to interleave M_1 and M_2 while gluing the initial and final moves. M_1 will be using the left segment of memory, while M_2 uses the right one. Consequently, we add N_1 when embedding EPS transitions from M_2.

ADD(0), DEL(0):

$$\frac{\dagger \xrightarrow{\mathsf{run}}{}_1 c_1^{(0)} \qquad \dagger \xrightarrow{\mathsf{run}}{}_2 c_2^{(0)}}{\dagger \xrightarrow{\mathsf{run}} (c_1^{(0)}, c_2^{(0)})} \qquad \frac{c_1^{(0)} \xrightarrow{\mathsf{done}}{}_1 \dagger \qquad c_2^{(0)} \xrightarrow{\mathsf{done}}{}_2 \dagger}{(c_1^{(0)}, c_2^{(0)}) \xrightarrow{\mathsf{done}} \dagger}$$

ADD(1):

$$\frac{c_1^{(0)} \xrightarrow{\ell}{}_1 (d_1^{(0)}, d_1^{(1)}) \qquad c \in C_2^{(0)}}{(c_1^{(0)}, c) \xrightarrow{\ell} ((d_1^{(0)}, c), d_1^{(1)})} \qquad \frac{c \in C_1^{(0)} \qquad c_2^{(0)} \xrightarrow{\ell}{}_2 (d_2^{(0)}, d_2^{(1)})}{(c, c_2^{(0)}) \xrightarrow{\ell} ((c, d_2^{(0)}), d_2^{(1)})}$$

DEL(1):

$$\frac{(c_1^{(0)}, c_1^{(1)}) \xrightarrow{\ell}{}_1 d_1^{(0)} \qquad c \in C_2^{(0)}}{((c_1^{(0)}, c_2^{(0)}), c) \xrightarrow{\ell} (d_1^{(0)}, c)} \qquad \frac{c \in C_1^{(0)} \qquad (c_2^{(0)}, c_2^{(1)}) \xrightarrow{\ell}{}_2 d_2^{(0)}}{((c, c_2^{(0)}), c_2^{(1)}) \xrightarrow{\ell} (c, d_2^{(0)})}$$

$\mathrm{ADD}(i), \mathrm{DEL}(i), \mathrm{EPS}(2j, 2i)$ $(i, j > 0)$ are simply copied over (in a way that respects state disjointness).

It remains to consider $\mathrm{EPS}(0, 2i)$.

$i > 0$ (where $h_1 = h,\ h_2 = N_1 + h$):

$$\frac{(0, h, v, c^{(2i)}) \xrightarrow{\epsilon}{}_j (v', d^{(2i)})}{(0, h_j, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})}$$

$i = 0$:

$$\frac{(0, h, v, c^{(0)}) \xrightarrow{\epsilon}{}_1 (v', d^{(0)}) \qquad c \in C_2^{(0)}}{(0, h, v, (c^{(0)}, c)) \xrightarrow{\epsilon} (v', (d^{(0)}, c))}$$

$$\frac{c \in C_1^{(0)} \qquad (0, h, v, c^{(0)}) \xrightarrow{\epsilon}{}_2 (v', d^{(0)})}{(0, N_1 + h, v, (c, c^{(0)})) \xrightarrow{\epsilon} (v', (c, d^{(0)}))}$$

Because the automaton can terminate only if both components reach states in which they can terminate, P1 follows from the inductive hypothesis. P2 is preserved because we can appeal to P2 for the relevant component and if moves from the other component interfere they can be postponed, because the automaton accepts the full interleaving.

$\mathbf{M \equiv M_1; M_2.}$

$$k = \mathsf{max}(k_1, k_2),\ N = N_1 + N_2 + 1,$$

$$C^{(0)} = (C_1^{(0)} \times I_{[\![\beta]\!]}) + C_2^{(0)},\ C^{(i)} = C_1^{(i)} + C_2^{(i)}\ (i > 0).$$

Here we need to run M_1 before M_2. There is a subtlety related to handling initial moves. The initial move needs to be remembered while the automaton for M_1 is running so that we can start M_2 with a suitable move. Recall that M_2 may be of any base type β.

As with parallel composition, M_1 uses the left segment of memory, and M_2 the right. We use the same strategy again when embedding EPS transitions. We also use an extra memory cell just to pass from M_1 to M_2. This is only because our definition of SA does not include a non-memory-modifying form of epsilon transition; this could be eliminated by introducing pure ϵ-transitions that do not manipulate memory.

ADD(0), DEL(0):

$$\frac{\dagger \xrightarrow{\mathsf{run}} {}_1c_1^{(0)} \qquad i \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{i} (c_1^{(0)}, i)} \qquad \frac{c_1^{(0)} \xrightarrow{\mathsf{done}} {}_1\dagger \qquad \dagger \xrightarrow{i} {}_2c_2^{(0)}}{(0, N, 0, (c_1^{(0)}, i)) \xrightarrow{\epsilon} c_2^{(0)}} \qquad \frac{c_2^{(0)} \xrightarrow{\mathsf{a}} {}_2\dagger}{c_2^{(0)} \xrightarrow{\mathsf{a}} \dagger}$$

Other transitions are defined in the following way.

- All transitions different from ADD(0), DEL(0) for M_1 are copied into the new automaton. Those referring to level-0 control states must additionally carry initial moves $I_{[\![\beta]\!]}$, which are copied during the transition.
- All transitions different from ADD(0), DEL(0) for M_2 are copied into the automaton. The only change required is the adjustment of memory operations: instead of $2j, h, v$ we need to use $2j, h + N_1, v$.

P1 holds since, on firing of the ε transition introduced by the second rule, we know there will be no children of the root thanks to P1 for M_1, as the firing conditions for the new rule correspond to the firing rules for done in M_1. It remains to observe that all children of M_2 are cleaned up by the time the a transition introduced by the third rule is fired, which follows from P1 for M_2 in the same way. P2 follows from the inductive hypothesis.

$\mathbf{M \equiv while\, M_1\, do\, M_2.}$

$$k = \mathsf{max}(k_1, k_2),\ N = N_1 + N_2 + 1,$$

$$C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ_0, \circ_1, \ldots, \circ_N\},\ C^{(i)} = C_1^{(i)} + C_2^{(i)}\ (i > 0).$$

We need to run M_1 and, depending on the final move, direct it to M_2 or terminate. When M_2 is about to finish, we redirect to M_1. Because we only delete the root at the very end, we need to re-initialise the memory explicitly on each iteration.

M_1 will be using the left segment of memory, M_2 works on the right one. As before, the extra reference is used to implement ϵ-transitions between M_1 and M_2.

ADD(0), DEL(0):

$$\frac{\dagger \xrightarrow{\mathsf{q}}{}_1 c_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} c_1^{(0)}} \qquad \frac{c_1^{(0)} \xrightarrow{0}{}_1 \dagger}{c_1^{(0)} \xrightarrow{\mathsf{done}} \dagger} \qquad \frac{c_1^{(0)} \xrightarrow{i}{}_1 \dagger \quad i>0 \quad \dagger \xrightarrow{\mathsf{run}}{}_2 c_2^{(0)}}{(0,N,0,c_1^{(0)}) \xrightarrow{\mathsf{done}} (0,c_2^{(0)})}$$

$$\frac{c_2^{(0)} \xrightarrow{\mathsf{done}}{}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{q}}{}_1 c_1^{(0)}}{\left\{\begin{array}{c} (0,N,0,c_1^{(0)}) \xrightarrow{\epsilon} (0,\circ_0) \qquad (0,j,0,\circ_{j-1}) \xrightarrow{\epsilon} (0,\circ_j)\ (1\le j\le N_1+N_2) \\ (0,N,0,\circ_{N_1+N_2}) \xrightarrow{\epsilon} (0,c_2^{(0)}) \end{array}\right\}}$$

Other transitions are copied across as per sequential composition above: all transitions except ADD(0), DEL(0) for M_1 are copied into the new automaton; likewise M_2 with the proviso that instead of $2j, h, v$ we need to use $2j, h+N_1, v$ in any EPS transitions.

The construction makes use of P1 for M_1: it was necessary to ensure that, when we move to M_2, the first one is really finished. To establish P1 for the resultant automaton we can appeal to P1 for M_2.

P2 follows from the fact that the resulting automaton accepts the concatention of traces from the two components.

$\mathbf{M} \equiv$ newvar x in $\mathbf{M_1}$. $k = k_1$, $N = N_1 + 1$, $C^{(i)} = C_1^{(i)}$ $(0 \le i \le k)$.

Here we add an extra cell at level 0 to store x. All transitions are copied over from M_1 except those labelled with x, which are handled as specified below.

Correctness follows from the fact that it suffices to restrict the work of M_1 to traces in which the relevant moves follow each other [70]. By P2, it suffices to consider scenarios in which the corresponding *transitions* follow each other.

Writing x:

$$\frac{c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}{}_1 (d^{(2i)}, d^{(2i+1)}) \xrightarrow{\mathsf{ok}^{(x,0)}}{}_1 e^{(2i)}}{(0,N,v,c^{(2i)}) \xrightarrow{\epsilon} (j,e^{(2i)}) \quad (0 \le v \le max)} \tag{9.1}$$

Reading x:

$$\frac{c^{(2i)} \xrightarrow{\mathsf{read}^{(x,\rho)}} {}_1(d^{(2i)}, d^{(2i+1)}) \xrightarrow{j^{(x,0)}} {}_1e^{(2i)}}{(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})} \tag{9.2}$$

Both P1 and P2 for the resultant automaton follow by appealing to the inductive hypothesis for the automaton for M_1.

$\mathbf{M} \equiv$ newsem s in $\mathbf{M_1}$. $k = k_1$, $N = N_1 + 1$, $C^{(i)} = C_1^{(i)}$ $(0 \le i \le k)$.

This is very similar to the previous case but the added transitions will both read and write in a single step.

Similarly, we add an extra cell at level 0 to store values of s. All transitions are copied over from M_1 except those labelled with s, which are handled as specified below.

Grabbing:

$$\frac{c^{(2i)} \xrightarrow{\mathsf{grb}^{(s,\rho)}} {}_1(d^{(2i)}, d^{(2i+1)}) \xrightarrow{\mathsf{ok}^{(s,0)}} {}_1e^{(2i)}}{(0, N, 0, c^{(2i)}) \xrightarrow{\epsilon} (1, e^{(2i)})}$$

Releasing:

$$\frac{c^{(2i)} \xrightarrow{\mathsf{rls}^{(s,\rho)}} {}_1(d^{(2i)}, d^{(2i+1)}) \xrightarrow{\mathsf{ok}^{(s,0)}} {}_1e^{(2i)}}{(0, N, 1c^{(2i)}) \xrightarrow{\epsilon} (0, e^{(2i)})}$$

$\mathbf{M} \equiv \mathbf{f}\ \mathbf{M_l} \cdots \mathbf{M_1}$. Suppose $\Gamma \vdash f M_l \cdots M_1 : \beta$ with $(f : \theta_l \to \cdots \to \theta_1 \to \beta) \in \Gamma$. Let $I_{[\![\beta]\!]}$ be the set of initial question-moves of $[\![\beta]\!]$. Given $q \in I_{[\![\beta]\!]}$, we write A_q for the set of corresponding answers.

We take $k = 2 + \mathsf{max}_{1 \le i \le l}\ k_i$, $N = \mathsf{max}_{1 \le i \le l}\ N_i$,

$$\begin{aligned} Q^{(0)} &= \{0_q, 1_q, 2_a \mid q \in I_{[\![\beta]\!]}, a \in A_q\}, \\ Q^{(1)} &= \{3_q \mid q \in I_{[\![\beta]\!]}\}, \\ Q^{(j+2)} &= \sum_{i=1}^{l} Q_i^{(j)} \quad (0 \le j \le k). \end{aligned}$$

First we add transitions corresponding to calling and returning from f:

$$\frac{q \in I_{[\![\beta]\!]} \qquad a \in A_q}{\dagger \xrightarrow{q} 0_q \qquad 0_q \xrightarrow{q^f} (1_q, 3_q) \qquad (1_q, 3_q) \xrightarrow{a^f} 2_a \qquad 2_a \xrightarrow{a} \dagger}$$

In state 3_q we want to enable the environment to spawn an unbounded number of copies of each of $\Gamma \vdash M_u : \theta_u$ $(1 \le u \le l)$. This is done through the following rules, which embed the actions of the automata for M_u while relabelling the moves. We use $\square_L, \square_R$ to refer to arbitrary left- and right-hand sides of transitions, which are to be copied by the rule.

- Moves from M_u corresponding to θ_u obtain an additional annotation fu, as they are now the uth argument of $f : \theta_l \to \cdots \to \theta_1 \to \mathbf{com}$.

$$\frac{\dagger \xrightarrow{m^{(\epsilon,0)}}{}_u c_u^{(0)}}{3_q \xrightarrow{m^{(fu,0)}} (3_q, c_u^{(0)})} \qquad \frac{c_u^{(0)} \xrightarrow{m^{(\epsilon,0)}}{}_u \dagger}{(3_q, c_u^{(0)}) \xrightarrow{m^{(fu,0)}} 3_q} \qquad \frac{\square_L \xrightarrow{m^{(\vec{i},\rho)}}{}_u \square_R \quad \vec{i} \neq \epsilon}{\square_L \xrightarrow{m^{(fu\vec{i},\rho)}} \square_R}$$

The pointer structure is simply inherited in this case, but an additional pointer needs to be created to q^f from formerly initial moves for M_u (i.e. $m^{(\epsilon,0)}$), which did not have a pointer earlier. Fortunately, because we also use $\rho = 0$ in initial moves to represent the lack of a pointer, by copying 0 now we indicate that the move $m^{(fu,0)}$ points one level up, i.e. at the q^f move, as required.

- Moves from M_u that originate from Γ, i.e. moves of the form $m^{(x_v\vec{i},\rho)}$, where $(x_v \in \theta_v) \in \Gamma$, need no relabelling except for question-moves $m^{(x_v,\rho)}$ that need to point at the initial move $m^{(\epsilon,0)}$. Leaving ρ unchanged in this case would mean pointing at $m^{(fu,0)}$, whereas we need to point at $m^{(\epsilon,0)}$ instead. To readjust such pointers, we simply add 2 to ρ in the relevant moves, and preserve ρ in other moves.

$$\frac{\square_L \xrightarrow{m^{(x_v,\rho)}}{}_u \square_R \qquad m \text{ is a question}}{\square_L \xrightarrow{m^{(x_v,\rho+2)}} \square_R}$$

$$\frac{\square_L \xrightarrow{m^{(x_v\vec{i},\rho)}}{}_u \square_R \qquad \vec{i} \neq \epsilon \text{ or } (\vec{i} = \epsilon \text{ and } m \text{ is an answer})}{\square_L \xrightarrow{m^{(x_v\vec{i},\rho)}} \square_R}$$

- Memory-related transitions are also included but because all states are now two levels deeper, we need to adjust the level that is being accessed by 2.

$$\frac{(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon}{}_u (v', d^{(2i)})}{(2j+2, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})}$$

After the construction P1 is satisfied, because we create only singleton nodes at levels 0 and 1, and the only way to arrive at the 0-level state 2^a is by deleting the unique node at level 1. So, when the automaton arrives at 2^a, the root is the only node.

P2 is satisfied by appealing to P2 for each M_u and noting that the construction implements all possible interleaving between them, so swapping transitions that come from different M_u leads to another trace.

$\mathbf{M} \equiv \lambda\mathbf{x}.\mathbf{M_1} : \theta_l \to \cdots \to \theta_1 \to \beta$**.** This case is dealt with simply by renaming labels in the automaton for $\Gamma, x : \theta_l \vdash M_1 : \theta_{l-1} \to \cdots \to \theta_1 \to \beta$: tags of the form $m^{(\vec{xi},\rho)}$ must be renamed as $m^{(\vec{li},\rho)}$. P1 and P2 are inherited.

$\mathbf{M} \equiv !\mathbf{M_1}$. Here we need to perform the same transitions as the automaton for M_1 would, if started from read. While doing so, read has to be relabelled as q. It suffices to copy all ADD(i), DEL(i) ($i > 0$) transitions as well as EPS transitions, with ADD(0) and DEL(0) transitions defined as follows.

$$\frac{\dagger \xrightarrow{\mathsf{read}} {}_1c^{(0)}}{\dagger \xrightarrow{\mathsf{q}} c^{(0)}}$$

P1 and P2 are inherited from M_1.

$\mathbf{M} \equiv \mathbf{grab}(\mathbf{M_1})$. Here we want to perform the same transitions as the automaton for M_1 would, if started from grb. At the same time, grb and the corresponding answer ok have to be relabelled as run and done respectively. As with the previous case, we copy all ADD(i), DEL(i) ($i > 0$) transitions as well as EPS transitions, with ADD(0) and DEL(0) transitions defined as follows.

$$\frac{\dagger \xrightarrow{\mathsf{grb}} {}_1c^{(0)}}{\dagger \xrightarrow{\mathsf{run}} c^{(0)}} \qquad \frac{c^{(0)} \xrightarrow{\mathsf{ok}} {}_1\dagger}{c^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

As previous, P1 and P2 are inherited from M_1.

$\mathbf{M} \equiv \mathbf{release}(\mathbf{M_1})$. This case is the same as the previous one, albeit substituting grb with rls.

$\mathbf{M} \equiv \mathbf{if}\,\mathbf{M_1}\,\mathbf{then}\,\mathbf{M_2}\,\mathbf{else}\,\mathbf{M_3}$. This case is similar to $M_1; M_2$. Once M_1 terminates, the automaton for either M_2 or M_3 must be activated, depending on the final move. The automaton for M_1 must remember the initial move and carry it. Below we give concrete rules for ADD(0) and DEL(0). All other transitions must be copied, except that transitions for M_1 must propagate the initial moves, and EPS transitions from M_2, M_3 must have their memory indices incremented by N_1.

$k = \mathsf{max}(k_1, k_2, k_3)$,

$N = N_1 + \mathsf{max}(N_2, N_3) + 1$,

$Q^{(0)} = (Q_1^{(0)} \times I) + Q_2^{(0)} + Q_3^{(0)}$,

$Q^{(i)} = Q_1^{(i)} + Q_2^{(i)} + Q_3^{(i)}$ $(0 < i \leq k)$.

and the newly induced transitions are as follows.

$$\frac{\dagger \xrightarrow{\mathsf{q}}{}_1 c^{(0)} \qquad x \in I}{\dagger \xrightarrow{x} (c^{(0)}, x)} \qquad \frac{c^{(0)} \xrightarrow{\mathsf{i}}{}_1 \dagger \qquad i > 0 \qquad \dagger \xrightarrow{x}{}_2 d^{(0)}}{(0, N, 0, c^{(0)}) \xrightarrow{\epsilon} (0, d^{(0)})}$$

$$\frac{c^{(0)} \xrightarrow{\mathsf{0}}{}_1 \dagger \qquad \dagger \xrightarrow{x}{}_3 d^{(0)}}{(0, N, 0, c^{(0)}) \xrightarrow{\epsilon} (0, d^{(0)})}$$

P1 and P2 follow from the inductive hypothesis.

$\mathbf{M \equiv M_1 := M_2}$. This case is also similar to $M_1; M_2$. First we direct the computation into the automaton for M_2 and, depending on the final move i, continue in the automaton for M_1 as if $\mathsf{write}(i)$ were played.
$k = \mathsf{max}(k_1, k_2)$, $N = N_1 + N_2 + 1$, $Q^{(i)} = Q_1^{(i)} + Q_2^{(i)}$ $(0 \leq i \leq k)$.

$$\frac{\dagger \xrightarrow{\mathsf{q}}{}_2 c^{(0)}}{\dagger \xrightarrow{\mathsf{run}} c^{(0)}} \qquad \frac{c^{(0)} \xrightarrow{\mathsf{i}}{}_2 \dagger \qquad \dagger \xrightarrow{\mathsf{write(i)}}{}_1 d^{(0)}}{(0, N, 0, c^{(0)}) \xrightarrow{\epsilon} (0, d^{(0)})} \qquad \frac{c^{(0)} \xrightarrow{\mathsf{ok}}{}_1 \dagger}{c^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

All transitions different from ADD(0) and DEL(0) need to copied (while respecting disjointness), again adding N_1 to the indices of memory cells for any EPS transitions from M_2. □

The proof above is superficially similar to the proof given of Theorem 7.2 over leafy automata, and indeed many of the constructions employ a similar structure. The main difference is in the reduced scope of ADD and DEL transitions, and the new usage of EPS to represent variables whose values were previously managed in control states. However, this shift in power in SA away from tree-based communication (still used in LLA) and towards shared memory informs an entirely new set of restrictions over FICA terms, which we shall now investigate.

9.4 Restricted-Semaphore FICA

Given the translation from FICA to SA above, we are able to observe and identify structural properties of the automata that are generated. The property which interests us most with regards to split automata is *idempotency* of transitions, to be defined shortly. In this section we shall present a new fragment of FICA for which certain problems (namely emptiness) are decidable, and show how this decidability arises through the lens of idempotency.

The new language fragment to be defined here is restricted-semaphore FICA, or *rs*FICA for short.

Definition 9.2. A FICA term $\Gamma \vdash M : \theta$ is *restricted-semaphore* if its β-normal, η-long form is such that subterms of the form $f\, M_l \cdots M_1$ $(l > 0)$ do not contain free variables of type **sem**. The language of all such FICA terms is *restricted-semaphore* FICA (*rs*FICA).

The restrictions imposed on FICA to produce *rs*FICA are relatively minimal. *rs*FICA retains all sequential features of FICA, such as unrestricted types, state, and looping constructs. It still even allows for some "shallow" use of semaphores.[3] For example, the term

(a) $\textbf{newsem}\, s\, \textbf{in}\, \textbf{grab}(s); f(\textbf{skip}); \textbf{release}(s)$

is restricted-semaphore, but

(b) $\textbf{newsem}\, s\, \textbf{in}\, f(\textbf{grab}(s))$

is not. Moreover, as an indication of the relative expressiveness of FICA compared to our previous restricted language LFICA, the interesting terms

(c) $\textbf{newvar}\, x := 0\, \textbf{in}\, \textbf{while}\, !x = 0\, \textbf{do}\, x := \mathit{succ}(!x)$

(d) $\textbf{newvar}\, x := 0\, \textbf{in}\, f(f(x := 1))$,

which violated the locality restriction of LFICA, are permitted in *rs*FICA. Restricted-semaphore FICA is not a strict improvement over local FICA though: term (b) above is local and so it is in LFICA but not *rs*FICA.

9.4.1 Restricted-semaphore split automata

Rather than introducing a new variant of split automata which structurally enforces the restricted-semaphore condition, instead we shall consider what restricted-semaphore FICA looks like within the context of the original model. We shall then use that as the basis for the decidability result to follow.

The first property we identify is one that we have seen before, namely *even-boundedness*. As with the translation of FICA to LA, our translation to SA maintains the property that nodes at even levels have an upper bound on the number of children they will have at any one time (*local even-boundedness*). However, unlike local LA, we do not maintain the stronger property that the *total* number of children of any even-level node across a run is bounded (*global even-boundedness*), as we cannot accommodate iteration while maintaining global even-boundedness. The local bound is derived directly from the syntax of the term; see Table 9.1 for how each construct affects the local bound.

[3]The reason for this is that in our construction shallow uses of semaphores always occur at the same level as its declaration, and so usage of the semaphore can be interpreted entirely through control states.

Term $\mathbf{M}$	Local bound $b(\mathbf{M})$
skip	1
i	1
$\mathbf{div}_\theta$	1
$\mathbf{op}(\mathbf{M_1})$	$b(\mathbf{M_1})$
$\mathbf{M_1}\|\|\mathbf{M_2}$	$b(\mathbf{M_1}) + b(\mathbf{M_2})$
$\mathbf{M_1}; \mathbf{M_2}$	$\mathsf{max}(\mathbf{M_1}, \mathbf{M_2})$
$\mathbf{while\, M_1\, do\, M_2}$	$\mathsf{max}(\mathbf{M_1}, \mathbf{M_2})$
$\mathbf{newvar\, x\, in\, M_1}$	$b(\mathbf{M_1})$
$\mathbf{newsem\, s\, in\, M_1}$	$b(\mathbf{M_1})$
$\mathbf{f\ M_l \cdots M_l}$	$\mathsf{max}(\mathbf{M_l}, \ldots, \mathbf{M_l})$
$\lambda\mathbf{x}.\mathbf{M_1}$	$b(\mathbf{M_1})$
$!\mathbf{M_1}$	$b(\mathbf{M_1})$
$\mathbf{grab}(\mathbf{M_1})$	$b(\mathbf{M_1})$
$\mathbf{release}(\mathbf{M_1})$	$b(\mathbf{M_1})$
$\mathbf{if\, M_1\, then\, M_2\, else\, M_3}$	$\mathsf{max}(\mathbf{M_1}, \mathbf{M_2}, \mathbf{M_3})$
$\mathbf{M_1} := \mathbf{M_2}$	$\mathsf{max}(\mathbf{M_1}, \mathbf{M_2})$

Table 9.1: The effect that each FICA construct has on the local bound of its corresponding split automaton at even levels. In this table $b(M)$ denotes the local bound derived from the syntax of the term M.

The second is the *restricted-semaphore* condition and it is specific to translation of the restricted-semaphore fragment of FICA. The restricted-semaphore property describes the constrained ways in which memory is manipulated by an automaton corresponding to an *rs*FICA term. Recall that memory is modified only through EPS transitions, which check the value of some memory cell and modify it atomically. This allows us to check whether a semaphore is free and then grab it immediately if so. This is in contrast to the read/write operations defined over non-semaphore variables, which either read the value of a memory cell without modification, or write to it without checking it.

Let us describe the restricted-semaphore condition in terms of the transitions of a split automaton. EPS transitions are of the form

$$(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)}).$$

Such a transition finds a data value d at level $2i$, labelled with control state $c^{(2i)}$; checks that the ancestor $2j$ levels above d has the value v in the h-th cell of its memory; and if so changes the value to v' and changes the control state at d to $d^{(2i)}$.[4] In this way, EPS transitions are completely atomic, reading and writing simultaneously.

By contrast, the transitions corresponding to reading and writing non-

[4]As with all previous models, if the preconditions are not met then the transition cannot be fired.

semaphore variables are as follows (cf. translation rules 9.1 and 9.2). This assumes the variables are not free, i.e. they have been bound by some **newvar** ... **in** construct.

Transitions for writing variables ($M := N$) are of this shape:

$$(0, N, v, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)}) \quad (0 \leq v \leq max).$$

Transitions for reading variables ($!M$) are of this shape:

$$(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)}).$$

Observe that the write transitions will fire regardless of the value held in the memory cell, since the same transition exists with v set to any possible value. Note also that the read transition always sets the value after the read to the same as it was before the read. In this sense, neither the reading nor writing transitions perform an atomic read/write operation despite making use of EPS transitions; and these are the only time such transitions appear in the translation except for semaphores. With this insight we are now ready to formalise the definition of the restricted-semaphore condition for SA.

Definition 9.3. A split automaton is *restricted-semaphore* if, for every transition

$$(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$$

with $v \neq v'$, there also exist transitions

$$(2j, h, v'', c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$$

for all $v'' \in V$.

Furthermore we can formally define the notion of shallowness with regards to the usage of a semaphore.

Definition 9.4. A semaphore s is *shallow* in a term $\Gamma \vdash \mathbf{newsem}\, s\, \mathbf{in}\, M : \theta$ if and only if there are no subterms M' of M of the form $f(N)$ such that s appears in N.

To borrow terminology from the previous chapter, we may say a semaphore is shallow if and only if its *applicative depth* is zero.

Uses of shallow semaphores may be adjusted to suit restricted-semaphore split automata in the following way. Suppose we are performing this adjustment while inductively translating *rs*FICA terms as per the proof of Theorem

9.1. A shallow use of a semaphore will initially be translated into

$$
\begin{aligned}
&(0, N, 0, c^{(0)}) \xrightarrow{\epsilon} (1, d^{(0)}) \quad (\mathsf{grb}) \\
\text{or } &(0, N, 1, c^{(0)}) \xrightarrow{\epsilon} (0, d^{(0)}) \quad (\mathsf{rls}).
\end{aligned}
$$

The memory cells and control states are at the same level, i.e. only one node, the node at level 0, is involved in such a transition. Such transitions can be simulated by using the level-0 control state as memory. Let us subscript by 1 the components of the original automaton derived from the translation. Then we take $C^{(0)} = C_1^{(0)} \times \{0, 1\}$, $C^{(i)} = C_1^{(i)}$ $(i > 0)$, and $N = N_1 + 1$. Transitions with access to the control state at the root must be modified accordingly. In all cases take j from $\{0, 1\}$.

ADD(0), DEL(0):

$$
\frac{\dagger \xrightarrow{\mathsf{q}} {}_1 d^{(0)}}{\dagger \xrightarrow{\mathsf{q}} (d^{(0)}, 0)} \qquad \frac{c^{(0)} \xrightarrow{\mathsf{i}} {}_1 \dagger}{(c^{(0)}, j) \xrightarrow{\mathsf{i}} \dagger}
$$

ADD(1), DEL(1):

$$
\frac{c^{(0)} \xrightarrow{\mathsf{q}} {}_1 (d^{(0)}, d^{(1)})}{(c^{(0)}, j) \xrightarrow{\mathsf{q}} ((d^{(0)}, j), d^{(1)})} \qquad \frac{(c^{(0)}, c^{(1)}) \xrightarrow{\mathsf{i}} {}_1 d^{(0)}}{((c^{(0)}, j), c^{(1)}) \xrightarrow{\mathsf{i}} (d^{(0)}, j)}
$$

EPS(0,0), excluding uses of the semaphore introduced by **newsem in**:

$$
\frac{(0, h, v, c^{(0)}) \xrightarrow{\epsilon} {}_1 (v', d^{(0)}) \quad h < N}{(0, h, v, (c^{(0)}, j)) \xrightarrow{\epsilon} (v', (d^{(0)}, j))}
$$

And finally the use of the semaphore itself is translated as follows.

$$
\frac{(0, N, l, c^{(0)}) \xrightarrow{\epsilon} {}_1 (l', d^{(0)})}{(0, N, 0, (c^{(0)}, l)) \xrightarrow{\epsilon} (0, (d^{(0)}, l'))} \tag{9.3}
$$

Note that the translation moves the value of the semaphore out of the memory cell and into the second component of the control state. We still use an epsilon transition, but the value of the memory cell accessed is always zero and so has no bearing on the operation of the transition. The antecedent of translation rule 9.3 does not fulfil the restricted-semaphore condition given in Definition 9.3, but the consequent does, and so the automaton resulting frm this translation is restricted-semaphore.

All transitions except those described above are copied directly from the derived automaton into the new one, which is now restricted-semaphore. Since the term we were translating is itself restricted-semaphore, the only dangerous occurrences of semaphores have been discharged as above.

Corollary 9.1. *For any given rs*FICA *term* $\Gamma \vdash M : \theta$*, we may construct a split automaton which is both locally bounded and restricted-semaphore, whose traces are the game-semantic plays of the* $\Gamma \vdash M : \theta$ *and whose accepting traces are the complete plays of* $\Gamma \vdash M : \theta$*.*

We shall now go on to show how to decide emptiness for split automata for which these two properties hold. In order to make this proof more digestible, we will reformulate the restricted-semaphore property in a more abstract way as a form of idempotency over transitions of the automaton.

Recall that, in some configuration of the automaton, each even-level data value is labelled by one control state and zero or more memory cells. Let us write $\langle m^{(2i)}, c^{(2i)} \rangle$ for the pair where the first component is a vector from V^N representing the stored memory mapped to by d, and the second component is the control state at d. In this scheme, split automaton EPS-transitions such as

$$(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$$

can be written as

$$(\langle m^{(2j)}[h \mapsto v], c^{(2j)} \rangle, \langle m^{(2i)}, c^{(2i)} \rangle) \xrightarrow{\epsilon} (\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, d^{(2i)} \rangle).$$

For restricted-semaphore split automata, we further get that

$$(\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, c^{(2i)} \rangle) \xrightarrow{\epsilon} (\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, d^{(2i)} \rangle).$$

Taking $q^{(2i)}$ and $r^{(2i)}$ to range over the combined memory-control states described above, we have discovered that the restricted-semaphore property implies the following property, which we call *idempotence*:

$$\begin{array}{ll} \text{if} & (q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}) \\ \text{then} & (r^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}). \end{array}$$

If we interpret the first transition as putting constraints on when $q^{(2i)}$ can be changed to $r^{(2i)}$, the idempotence property mandates that if this change can be done in one node of the configuration tree, it can be done in an arbitrary number of nodes, and arbitrarily many times. This property is crucial for the decidability argument in the next section.

9.5 Idempotent Automata

We shall now prove that the emptiness problem is decidable for split automata which are both locally bounded and idempotent, and hence that a class of verification problems for rsFICA is decidable.

The decidability result will not be proven directly over split automata, but over a slightly more abstract model which makes use of similar notions to the combined transitions presented above. The memory-control division was instrumental to expressing properties of the translation of rsFICA to automata. For the decidability proof, all that matters is that the local-boundedness and idempotency properties hold, and so we do not need to work in the split setting.

We shall work again with pairs $\langle m^{(2i)}, c^{(2i)} \rangle$ representing the combination of memory cells and control states, respectively, associated to some data value. Recall that the mapping from data value to vector of memory cells is partial. We will take the empty vector when the image of a data value is undefined. In particular, nodes at odd levels will have an empty first component, and at even levels a vector of length N. $q^{(i)}$ and $r^{(i)}$ will continue to be used for such combined states.

We will now define *idempotent automata*. These are functionally the same as split automata with the exception that control and memory are taken together, and the idempotence and local-boundedness properties are enforced.

Definition 9.5. An *idempotent automaton* is a tuple $\langle Q, k, \Sigma, \delta \rangle$, where:

- k is the depth parameter;
- $Q = \bigcup \{Q^{(i)} : i = 0, \ldots, k\}$ is a finite set of states;
- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is the alphabet; and
- δ contains transitions of the shape given below, taking $i > 0$, $q \in \Sigma_{\mathsf{Q}}$, $a \in \Sigma_{\mathsf{A}}$, and $q^{(i)}, r^{(i)} \in Q^{(i)}$.

$$
\begin{array}{rcl}
\mathrm{ADD}(0) & \dagger \xrightarrow{q} r^{(0)} & \\
\mathrm{DEL}(0) & q^{(0)} \xrightarrow{a} \dagger & \\
\mathrm{ADD}(2i) & q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)}) & \\
\mathrm{DEL}(2i) & (q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} q^{(2i-1)} & \\
\mathrm{ADD}(2i+1) & q^{(2i)} \xrightarrow{q} (r^{(2i)}, r^{(2i+1)}) & \\
\mathrm{DEL}(2i+1) & (q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} r^{(2i)} & \\
\mathrm{EPS}(2j, 2i) & (q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}) & j \leq i
\end{array}
$$

We require two conditions:

- Idempotence:

$$\text{For every transition } (q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})$$
$$\text{We have } (r^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)});$$

- Local boundedness: there exists some (minimum) bound B such that, in every configuration reachable from the initial configuration κ_0, every even-level node has at most B children.

With memory and control states combined in idempotent automata, configurations become triples (D, E, f) where f is now a function from data values to combined states $\langle m, c\rangle$. The initial configuration is $\kappa_0 = (\emptyset, \emptyset, \emptyset)$.

The definitions of a run and acceptance are as for split automata. It should be clear that restricted-semaphore, locally-bounded split automata can be simulated by idempotent automata by using the combined states in idempotent automata to represent control states combined with memory. As discussed in the previous section, the bound B in the local boundedness condition can be derived from the syntax of the term. Observe that, due to the shape of transitions, the states at odd levels never change. (As an aside, the local boundedness condition is formulated as a semantic property, but it could be enforced in the syntax—and hence in the structure of the automaton—by maintaining a count of children at each node as part of that node's control state, with ADD transitions on children of the node incrementing that counter and corresponding DEL transitions decrementing it.)

We interest ourselves with the emptiness problem over idempotent automata (and hence over locally-bounded restricted-semaphore split automata): does a given idempotent automaton accept any word? We show the decidability of this problem via a lemma which allows us to reduce the depth of such an automaton. The lemma effectively eliminates two levels of data values. By repeatedly applying the lemma we will eventually reach an automaton with depth parameter $k \leq 2$, whose emptiness problem reduces to emptiness of standard finite automata.

Lemma 9.1. *For every idempotent automaton $\mathcal{A}$ with $2i+2$ levels, one can construct an idempotent automaton $\mathcal{A}^{\uparrow}$ with $2i$ levels such that the language of $\mathcal{A}$ is nonempty if and only if the language of $\mathcal{A}^{\uparrow}$ is nonempty.*

Proof. Take B to be an upper bound on the number of children a node at an even level can have. The bound comes from the local boundedness property of $\mathcal{A}$. The states of $\mathcal{A}^{\uparrow}$ are the states of $\mathcal{A}$, except for at level $2i$:

$$Q^{\uparrow(2i)} = Q^{(2i)} \times (\{1, \ldots, B\} \rightharpoonup (Q^{(2i+1)} \times \mathcal{P}(Q^{(2i+2)})))$$

where the second argument is the set of partial functions as displayed. We write $\emptyset$ for the empty function, and $g(i) = \bot$ when g is not defined at i.

The intention is to supplement the state at level $2i$ with a second component, describing the subtrees rooted at level $2i + 1$ that are eliminated by the construction. A node at level $2i$ can have at most B children, hence the second component is a partial function with domain $\{1, \ldots, B\}$. The values of this function are representations of subtrees rooted at the children of the node: the label of a node at level $2i + 1$, and the set of labels of children of this node. The representation loses information about the precise number of children with each label. Consequently, there are fintely many representations of subtrees at level $2i + 1$.

For a configuration (D, E, f) and a data value $d \in E$, we write $\overline{fE}(d)$ for the labelled subtree rooted at d. A pair $(q^{(2i+1)}, S) \in Q^{(2i+1)} \times \mathcal{P}(Q^{(2i+2)})$ represents a subtree rooted at a node of level $2i + 1$: the root is labelled by $q^{(2i+1)}$, and the set of labels of the children of the root is S. We write $\mathit{setrep}(\overline{fE}(d^{(2i+1)}))$ for this representation of the subtree rooted at $d^{(2i+1)}$.

A state $(q^{(2i)}, g) \in Q^{\uparrow(2i)}$ represents a subtree rooted at a node of level $2i$ with $q^{(2i)}$ the label of the node, and g representing at most B subtrees rooted at the children of the node as described above.

The transitions of $\mathcal{A}^{\uparrow}$ reflect this representation of subtrees by states. The way to modify the transitions is presented in Figure 9.1. We continue with the same treatment of translation rules: the antecedent (above the line) existing in $\mathcal{A}$ implies the existence of the consequent (below the line) in $\mathcal{A}^{\uparrow}$. Observe that only transitions involving levels $2i$, $2i + 1$, and $2i + 2$ are modified; all others are copied directly.

The intuition behind these rules follows the intuition behind the definition of $Q^{\uparrow(2i)}$ we saw earlier. For example, consider the first rule, ADD$(2i)$. This creates a new node at level $2i$ labelled by $r^{(2i)}$. The rule is changed to an ADD$(2i)$ rule that creates a new node labelled by $(r^{(2i)}, \emptyset)$, where $\emptyset$ is the empty function representing that the newly-created node has no children.

Another example is the last rule, EPS$(2i{+}2, 2i{+}2)$. This form of transition in automaton $\mathcal{A}$ changes the state at a node at level $2i{+}2$. This is replaced by the change of one of the elements in the set S. The notation we use actually describes two possible ways to instantiate the rule, depending on the runtime configuration of the automaton:

- There may be no $q^{(2i+2)}$ in S after firing the transition. In this case, $q^{(2i+2)}$ is removed and replaced by $r^{(2i+2)}$ in S.
- There may still be $q^{(2i+2)}$ remaining in S. The transition adds $r^{(2i+2)}$ to

ADD($2i$):

$$\frac{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)})}{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, (r^{(2i)}, \emptyset))}$$

DEL($2i$):

$$\frac{(q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} q^{(2i-1)}}{(q^{(2i-1)}, (q^{(2i)}, \emptyset)) \xrightarrow{a} q^{(2i-1)}}$$

EPS($2j, 2i$):

$$\frac{(q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})}{(q^{(2j)}, (q^{(2i)}, g)) \xrightarrow{\epsilon} (r^{(2j)}, (r^{(2i)}, g))}$$

ADD($2i+1$):

$$\frac{q^{(2i)} \xrightarrow{q} (r^{(2i)}, r^{(2i+1)})}{(q^{(2i)}, g[l \mapsto \bot]) \xrightarrow{q} (r^{(2i)}, g[l \mapsto (r^{(2i+1)}, \emptyset)])}$$

DEL($2i+1$):

$$\frac{(q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} r^{(2i)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, \emptyset)]) \xrightarrow{a} (r^{(2i)}, g[l \mapsto \bot]))}$$

ADD($2i+2$):

$$\frac{q^{(2i+1)} \xrightarrow{q} (q^{(2i+1)}, r^{(2i+2)})}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S)]) \xrightarrow{q} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

DEL($2i+2$):

$$\frac{(q^{(2i+1)}, q^{(2i+2)}) \xrightarrow{a} q^{(2i+1)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{a} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S)])}$$

EPS($2j, 2i+2$):

$$\frac{(q^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)}) \qquad 2j < 2i}{(q^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})])) \xrightarrow{\epsilon} (r^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})]))}$$

EPS($2i, 2i+2$):

$$\frac{(q^{(2i)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2i)}, r^{(2i+2)})}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{\epsilon} (r^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

EPS($2i+2, 2i+2$):

$$\frac{q^{(2i+2)} \xrightarrow{\epsilon} r^{(2i+2)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{\epsilon} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

Figure 9.1: Translation rules for building an idempotent automaton $\mathcal{A}^{\uparrow}$ of depth $2i$ based on one of depth $2i+2$.

S without removing $q^{(2i+2)}$.

The two ways are useful: the first corresponds to a situation when there is exactly one child labelled by $q^{(2i+2)}$, the second when there is more than one.

Observe that all transitions affecting levels $(2i+1)$ and $(2i+2)$ are translated to EPS$(2i, 2i)$ transitions, except for EPS$(2j, 2i+2)$ for $2j < 2i+2$. The idempotence property of the resulting automaton $\mathcal{A}^{\uparrow}$ follows from the idempotence of $\mathcal{A}$.

We need to show that $\mathcal{A}$ accepts some data word if and only if $\mathcal{A}^{\uparrow}$ accepts some, possibly different, data word. The two data words will be different when the one accepted by $\mathcal{A}$ uses data at levels below $2i$; these are not accessible for $\mathcal{A}^{\uparrow}$.

For the proof we introduce a concept of *indexed runs* of $\mathcal{A}$. In practice this means we add a fourth component to configurations that assigns numbers to some nodes. An indexed configuration is (D, E, f, ind) where $\mathit{ind} : E \rightharpoonup \{1, \dots, B\}$ is a partial function defined for all data of level $2i+1$ in E. Intuitively, *ind* gives unique identifiers to siblings at level $2i+1$. When a new node at level $2i+1$ is created, it gets the smallest index different from the indices of its siblings. The node keeps this index until the node is removed by a DEL transition. Since a node at level $2i$ can have at most B children at once, we have enough indices to uniquely map each contemporaneous child to a unique number. An accepting indexed run has the form:

$$(\emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{b_1} \cdots (D_l, E_l, f_l, \mathit{ind}_l) \xrightarrow{b_l} \cdots (\emptyset, \emptyset, \emptyset, \emptyset)$$

where as before every b_l is either ϵ or a data letter (t_l, d_l) consisting of a letter $t_l \in \Sigma$ and a data value $d_l \in \mathcal{D}$. The indexing functions allow us to define $\mathit{state}(f_l\bar{E}_l(d^{(2i)}), \mathit{ind}_l) \in Q^{\uparrow(2i)}$ for every $d^{(2i)} \in E_l$:

$$\mathit{state}(f_l\bar{E}_l(d^{(2i)}), \mathit{ind}_l) = (f_l(d^{(2i)}), g)$$

where $g(\mathit{ind}_l(d^{(2i+1)})) = \mathit{setrep}(f_l\bar{E}_l(d^{(2i+1)}))$ for every child $d^{(2i+1)}$ of $d^{(2i)}$ in E_l.

From an indexed run ν of $\mathcal{A}$, we construct a run of $\mathcal{A}^{\uparrow}$ by induction. Suppose that we have constructed a run $\mu^{\uparrow}$ of $\mathcal{A}^{\uparrow}$ corresponding to a prefix μ of the run of $\mathcal{A}$. Run $\mu^{\uparrow}$ reaches a configuration $(D^{\uparrow}, E^{\uparrow}, f^{\uparrow})$ while run μ reaches (D, E, f, ind). We assume that the following invariants hold:

(1) When restricted to levels $\leq 2i$, set $D^{\uparrow}$ is the same as D, and $E^{\uparrow}$ is the same as E.

(2) For every $d \in E^{\uparrow}$ of level $< 2i$ we have $f^{\uparrow}(d) = f(d)$.

(3) For every $d^{(2i)} \in E^{\uparrow}$ (of level $2i$), we have $f^{\uparrow}(d^{(2i)}) = \mathit{state}(\overline{fE}(d^{(2i)}), \mathit{ind})$.

We consider the next transition on the run ν. If it does not concern level $2i$ or below then the same transition can be executed by $\mathcal{A}^{\uparrow}$ without modification. If it does we examine the possible cases transition of $\mathcal{A}^{\uparrow}$ preserves the invariants. This way we prolong μ and $\mu^{\uparrow}$ while keeping the invariants. Property (1) implies that $\mu^{\uparrow}$ is accepting if μ is.

For the other direction we consider an accepting run $\nu^{\uparrow}$ of $\mathcal{A}^{\uparrow}$. Let ℓ be the length of $\nu^{\uparrow}$. By induction, for every prefix $\mu^{\uparrow}$ of this run we construct a run μ satisfying the same invariant as above, and moreover:

(4) Every node of level $2i+2$ in (D, E, F) has at least $2^{\ell-|\mu^{\uparrow}|}$ siblings with the same label.

The construction of μ is by cases depending on the type of transitions in the run of $\mathcal{A}^{\uparrow}$. We need sufficiently big multiplicities of leaves to simulate set operations where the state on the left hand-side does not disappear (cf. our discussion above concerning EPS$(2i+2, 2i+2)$) rule). We can get arbitrary multiplicities thanks to the idempotency of the rules.

Let us examine a representative case of the EPS$(2j, 2i+2)$ rule for $2j < 2i$. Suppose $\mathcal{A}^{\uparrow}$ applies the following transition at node $d^{(2i)}$:

$$(q^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})])) \\ \xrightarrow{\epsilon} \\ (r^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})]))$$

By the invariant $f^{\uparrow}(d^{(2i)}) = \mathit{state}(\overline{fE}(d^{(2i)}), \mathit{ind})$, consider $d^{(2i+1)}$ with $\mathit{ind}(d^{(2i+1)}) = l$. We have that it has a child labeled $q^{(2i+2)}$. By invariant (4), it has at least $2^{\ell-|\mu^{\uparrow}|}$ children labeled $q^{(2i+2)}$. On the side of automaton $\mathcal{A}$ we can then fire the corresponding EPS$(2j, 2i+2)$ transition

$$(q^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)}),$$

followed by some number of transitions

$$(r^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)}).$$

If $q^{(2i+2)}$ does not appear in $S \cup \{r^{(2i+2)}\}$ then the above rule is used to change all occurrences of $q^{(2i+2)}$ below $d^{(2i+1)}$ to $r^{(2i+2)}$. If it does appear then $2^{(\ell-|\mu^{\uparrow}|)-1}$ occurrences are changed, leaving the rest. This reestablishes the fourth invariant both for $q^{(2i+2)}$ and $r^{(2i+2)}$. Observe that if invariant (4) talked about $(\ell - |\mu^{\uparrow}|)$ siblings instead of $2^{\ell-|\mu^{\uparrow}|}$ then it would not be possible to reestablish it at this point. □

Repeated applications of Lemma 9.1 reduce the emptiness problem of an idempotent automaton to the emptiness problem of a standard finite automaton. Indeed, an idempotent automaton whose depth parameter is 0 is just a finite automaton. Hence:

Theorem 9.2. *Emptiness of idempotent automata is decidable.*

Idempotent automata exhibit the same behaviours as split automata, except that the memory and control components are no longer split (since this property is not required for our purposes once idempotence of transitions has been shown). It follows that we can reduce the emptiness problem on restricted-semaphore split automata to the emptiness problem on idempotent automata. Hence, since we can (per Theorem 9.1) construct a restricted-semaphore split automaton whose language faithfully represents complete plays of a given *rs*`FICA` term, we get that

Theorem 9.3. *It is decidable whether, given an rs*`FICA` *term* $\Gamma \vdash M : \theta$*, there exists a context* $\mathcal{C}$ *such that*

- $\vdash \mathcal{C}[M] : \mathbf{com}$*; and*
- $\mathcal{C}[M]$ *may terminate by some computation in which* $\mathcal{C}$ *uses* M*.*

9.6 Stuttering Invariant Properties

With may-termination of *rs*`FICA` terms proven to be decidable using the above construction, it remains to ask: what can we do with *rs*`FICA`? In this section we endeavour to provide some meaningful examples of properties that can be verified in the *rs*`FICA` language.

As we have seen, it is possible to translate any *rs*`FICA` term into an automaton whose language faithfully represents the set of complete plays in the game semantics for the given term. The automaton operates over a data tree of some depth, which reflects the syntactic structure of the underpinning λ-term. We fix some level $2i$ of the data tree, and for every node at level $2i$ we look at the sequence of control states visited by the node during a run. We check if some, or every, such sequence satisfies some given regular property. Later we shall see how to express certain common classes of program analysis properties in this way.

We first describe automata constructions, and then give some applications. We try to give an idea of the constructions without going to excessive details that are not difficult but tedious. To fix the notation, consider an idempotent automaton $\mathcal{A}$, and a level $2i$ of the dataset. We assume that we are given a standard finite automaton $\mathcal{C}$ defining interesting sequences of states at level $2i$. So the alphabet of $\mathcal{C}$ is $Q^{(2i)}$, i.e. the set of states of $\mathcal{A}$ at level $2i$. We use q^c to refer to states of $\mathcal{C}$. The initial state of $\mathcal{C}$ is q^c_{init}, and the set of accepting states of $\mathcal{C}$ is $Fin^{\mathcal{C}}$.

We want to check if there is an accepting run of $\mathcal{A}$ such that every node at level $2i$ goes through a sequence of states accepted by $\mathcal{C}$. In other words, if there is an accepting run such that for every data value d of level $2i$ appearing in the run the sequence of states that label d is accepted by $\mathcal{C}$. If this holds, we say that there is an accepting run where every level-$2i$ sub-run satisfies $\mathcal{C}$.

We convert an idempotent automaton $\mathcal{A}$ into an idempotent automaton $\mathcal{A}^{\forall\mathcal{C}}$ such that: $\mathcal{A}^{\forall\mathcal{C}}$ has an accepting run if and only if $\mathcal{A}$ has an accepting run where every level-$2i$ sub-run satisfies $\mathcal{C}$. This reduces the question to the emptiness of idempotent automata that, as we have seen, is decidable.

For this construction to work we require that automaton $\mathcal{C}$ describes a property that is closed with respect to stuttering:

Definition 9.6. An automaton $\mathcal{C}$ is *stuttering-invariant* if and only if, for every b in the alphabet of $\mathcal{C}$,

$$w_1 b w_2 \in L(\mathcal{C}) \iff w_1 b b w_2 \in L(\mathcal{C}).$$

We assume in what follows that $\mathcal{C}$ is a minimal deterministic automaton. Observe that if $\mathcal{C}$ is stuttering-invariant then, for every transition $q^c_1 \xrightarrow{b} q^c_2$, it has also the transition $q^c_2 \xrightarrow{b} q^c_2$. (In particular, this will only hold in the minimal automaton, and only when the automaton is deterministic).

To construct $\mathcal{A}^{\forall\mathcal{C}}$ we modify the the states at level $2i$ of $\mathcal{A}$, and transitions involving this level. The new set of sates at level $2i$ is $Q^{(2i)} \times Q^c$—we add states of $\mathcal{C}$ as an additional component.

Transitions are modified in the following way:

$$\text{ADD}(2i) \quad \frac{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)})}{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, (r^{(2i)}, q^c))} \qquad \text{if } q^c_{init} \xrightarrow{r^{(2i)}} q^c$$

$$\text{DEL}(2i) \quad \frac{(q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} r^{(2i-1)}}{(q^{(2i-1)}, (q^{(2i)}, q^c)) \xrightarrow{a} q^{(2i-1)}} \qquad \text{if } q^c \in Fin^{\mathcal{C}}$$

ADD($2i$) transitions initialise the $\mathcal{C}$ component. DEL($2i$) transitions allow the removal of a level-$2i$ node provided the $\mathcal{C}$ component is in an accepting state.

We modify other transitions involving level $2i$ so that the $\mathcal{C}$ component is updated as expected. Supposing $q^c_1 \xrightarrow{r^{(2i)}} q^c_2$ is a transition of $\mathcal{C}$:

$$\text{EPS}(2j,2i) \quad \frac{(q^{(2j)},q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)},r^{(2i)})}{(q^{(2j)},(q^{(2i)},q^c_1)) \xrightarrow{\epsilon} (r^{(2j)},(r^{(2i)},q^c_2))}$$

$$\text{ADD}(2i+1) \quad \frac{q^{(2i)} \xrightarrow{q} (r^{(2i)},r^{(2i+1)})}{(q^{(2i)},q^c_1) \xrightarrow{q} ((r^{(2i)},q^c_2),r^{(2i+1)})}$$

$$\text{DEL}(2i+1) \quad \frac{(q^{(2i)},q^{(2i+1)}) \xrightarrow{a} r^{(2i)}}{((q^{(2i)},q^c_1),q^{(2i+1)}) \xrightarrow{a} (r^{(2i)},q^c_2)}$$

$$\text{EPS}(2i,2j) \quad \frac{(q^{(2i)},q^{(2j)}) \xrightarrow{\epsilon} (r^{(2i)},r^{(2j)})}{((q^{(2i)},q^c_1),q^{(2j)}) \xrightarrow{\epsilon} ((r^{(2i)},q^c_2),r^{(2j)})}$$

Observe that we have two types of EPS transitions to handle, depending on whether $2i$ is the lower or the upper level. Note that EPS transitions of the first kind are idempotent in $\mathcal{A}^{\forall\mathcal{C}}$ if they were in $\mathcal{A}$. Similarly for the second kind, but here additionally we rely on stuttering-invariance of $\mathcal{C}$.

Lemma 9.2. *If $\mathcal{A}$ is an idempotent automaton and $\mathcal{C}$ is a stuttering-invariant minimal deterministic automaton then $\mathcal{A}^{\forall\mathcal{C}}$ is an idempotent automaton. Moreover, $\mathcal{A}$ has an accepting run whose level-$2i$ sub-runs all satisfy $\mathcal{C}$ if and only if $\mathcal{A}^{\forall\mathcal{C}}$ has an accepting run.*

We can modify the above construction to answer a different question: is there any accepting run with at least one $2i$ sub-run satisfying $\mathcal{C}$? Automaton $\mathcal{A}^{\exists\mathcal{C}}$ for this question is constructed from $\mathcal{A}^{\forall\mathcal{C}}$. The idea is that once one of the sub-runs at level $2i$ reaches an accepting state, it stores this fact in the root state. At the same time we should ensure that this sub-run terminates after this action. In the universal case we could use termination detection to collect acceptance information: in an accepting run all components at level $2i$ needed to disappear, and they could disappear only when they reached an accepting state. In the existential case we need to implement an ad-hoc mechanism for terminating the automaton once a suitable sub-run has been observed anywhere in the machine.

In $\mathcal{A}^{\exists\mathcal{C}}$, the set of states at level 0 becomes $Q^{(0)} \times \{tt, ff\}$; the second component indicating if there has yet been a $2i$ sub-run satisfying $\mathcal{C}$. The modification of transitions reflects this intuition:

$$\text{ADD}(0)\frac{\dagger \xrightarrow{q} r^{(0)}}{\dagger \xrightarrow{q} (r^{(0)},ff)} \qquad \text{DEL}(0)\frac{q^{(0)} \xrightarrow{a} \dagger}{(q^{(0)},tt) \xrightarrow{a} \dagger}$$

The ADD transition says that we initialize the second component to *ff*, the DEL transition says that an accepting run should end with the second component set to *tt*.

For other transitions of $\mathcal{A}^{\exists\mathcal{C}}$, we take transitions of $\mathcal{A}^{\forall\mathcal{C}}$ with some modifications. Transitions involving level 0 are modified so that they leave the second component unchanged. The second component at the root can be changed only by a new ϵ-transition at level $2i$ that we introduce now. In $\mathcal{A}^{\exists\mathcal{C}}$ the set of states at level $2i$ is $Q^{(2i)} \cup (Q^{(2i)} \times Q^c)$. So a state at level $2i$ of $\mathcal{A}^{\exists\mathcal{C}}$ is either a state of $\mathcal{A}^{\forall\mathcal{C}}$ or a state of $\mathcal{A}$ at this level. The idea is that at the end of a sub-computation simulated at some node at level $2i$, the automaton can set the component at level 0 to *tt* if the computation finished in an accepting state of $\mathcal{C}$. We have the following for arbitrary states and arbitrary $\alpha \in \{tt, ff\}$.

$$\begin{aligned}
&\mathrm{EPS}(0,2i)\colon\quad ((q^{(0)},\alpha),(q^{(2i)},q^c)) \xrightarrow{\epsilon} ((q^{(0)},tt),q^{(2i)}) \quad \text{if } q^c \in F\\
&\mathrm{EPS}(0,2i)\colon\quad ((q^{(0)},\alpha),(q^{(2i)},q^c)) \xrightarrow{\epsilon} ((q^{(0)},\alpha),q^{(2i)})
\end{aligned}$$

It is clear that these transitions are idempotent. The other transitions are as in automaton $\mathcal{A}^{\forall\mathcal{C}}$ except for the transitions of type DEL$(2i)$ which are instead copied from $\mathcal{A}$:

$$\mathrm{DEL}(2i): \quad (q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} r^{(2i-1)}$$

The idea behind this is that DEL$(2i)$ can be performed only after one of the above EPS$(0,2i)$ transitions has fired, which in turn can only occur once some accepting sub-run at level $2i$ has been witnessed.

Lemma 9.3. *If $\mathcal{A}$ is an idempotent automaton and $\mathcal{C}$ is a stuttering-invariant minimal deterministic automaton then $\mathcal{A}^{\exists\mathcal{C}}$ is an idempotent automaton. Moreover, $\mathcal{A}$ has an accepting run whose some level-$2i$ sub-run satisfies $\mathcal{C}$ if and only if $\mathcal{A}^{\exists\mathcal{C}}$ has an accepting run.*

Proposition 9.1. *The following questions are decidable, for idempotent automaton $\mathcal{A}$, data level $2i$, and stuttering-invariant finite automaton $\mathcal{C}$:*

(1) Is there an accepting run of $\mathcal{A}$, all level-$2i$ sub-runs of which satisfy $\mathcal{C}$?

(2) Is there an accepting run of $\mathcal{A}$, some level-$2i$ sub-run of which satisfies $\mathcal{C}$?

(3) Do all accepting runs of $\mathcal{A}$ have some level-$2i$ sub-run satisfying $\mathcal{C}$?

(4) Do all accepting runs of $\mathcal{A}$ have all level-$2i$ sub-runs satisfying $\mathcal{C}$?

Questions (1) and (2) are solved directly by the two lemmas above. The remaining two are obtained by considering the dual question for an automaton whose language is the complement of $\mathcal{A}$. Observe that the complement of a stuttering invariant language is also stuttering invariant. In fact, this is the only point where we need the full stuttering invariance of $\mathcal{C}$—in all of the arguments until this point, we have required only that $\mathcal{C}$ is closed under stuttering expansion.

This proposition allows the verification of *rs*`FICA` terms to be reduced to emptiness checking of idempotent automata. Given a term we use the translation from the proof of Theorem 9.1 to obtain a split automaton. This automaton is an idempotent automaton by Corollary 9.1. We can then take an automaton $\mathcal{C}$ expressing a property of interest. With stuttering invariant finite automata $\mathcal{C}$ we can express such properties as:

- Is a given variable ever set to a given value?
- Is a given variable invariant in a loop?
- Is a given variable ever read after being set?

As a side note, recall that in *rs*`FICA` reads are silent: reads are simulated by epsilon-transitions in split automata. As a result, reads do not directly appear in traces. In order to check the last property above, therefore, some additional gadgetry will need to be used to ensure that the read to be checked has some visible side-effect. For example, we could instrument a term such that whenever such a read occurs, some visible effect is triggered (for example some function is called or some command is executed).

Proposition 9.1 gives a method to verify these properties for all quantification combinations: every/some accepting run ν, and every/some sub-run of ν. Indeed, the proposition effectively reduces verification of all these questions to emptiness checking of a suitable idempotent automaton. The latter is decidable by Theorem 9.2.

9.7 Remarks on *rs*`FICA`

As remarked previously, restricted-semaphore `FICA` is significantly more expressive than `LFICA` in a number of ways. For example, the ability to represent unbounded iteration and arbitrarily nested function calls in *rs*`FICA` makes it capable of expressing a much larger fragment of the `FICA` language's constructs—indeed, all sequential constructs of `FICA` can appear in *rs*`FICA`.

However, the restriction on the use of semaphores corresponds to removing unbounded *synchronisation* in *rs*`FICA` terms. This is a limitation on the ability to express more complex parallel programs. `LFICA` is only slightly better in this regard: while unbounded use of semaphores can occur in *rs*`FICA`, no data (variables or semaphores) may traverse an applicative depth (nested calls to free functions) of more than 1, which places an implicit upper bound on the amount of usefulness that any such semaphore can offer.

By relating our model to the game-semantic view of the idealized concurrent ALGOL we are able to verify properties in arbitrary contexts. The ability to check properties using arbitrary quantification is a particular strength of this approach. It is of particular note that while recursion is not permitted in *rs*FICA, we do still permit unbounded iteration—so there is a striking difference between the relative power of iteration and recursion in this setting.

In this work we have primarily focused on the emptiness problem for these automata, as that alone is enough to empower checking of a large class of verification properties. The complexity of other decision problems for split automata (restricted-semaphore or otherwise) remains uninvestigated. In particular, we believe that it is worth investigating the potential of decidability of equivalence for the mode. If decidable, this could be used for testing contextual equivalence for *rs*FICA terms. (However the encoding of terms into plays is not injective—several different plays may represent the same term. As such, checking equivalence directly on representations of plays may produce false negatives. Additional machinery would need to be devised to deal with this.)

As a closing remark, while we proved that emptiness for idempotent automata is decidable, we did not wrestle with complexity bounds for the problem. The procedure as given in this work has complexity equivalent to a tower of exponentials whose height is dependent on the depth parameter of the idempotent automaton in question (and hence, when verifying properties over *rs*FICA terms, to the applicative depth of the given term). We believe that there may be scope to reduce this complexity—doing so would be of value if idempotent automata are to be used for practical verification.

Chapter 10

Closing

This work respresents the culmination of five years of ongoing research in concert with a number of esteemed research colleagues. We shall close out the thesis by revisiting the various contributions made, and along the way identifing further avenues of research that extend those contributions. We shall also lightly touch on some other work that the author has undertaken to complete, each of which relates to some of the content from Chapters 4 to 9.

10.1 Petri Net Coverability

Depending on one's accounting, this work includes three distinct complete implementations of decision procedures for the general Petri net (equivalently `VASS`) coverability problem: the Karp-Miller tree derivation procedure from Section 4.3.1 implemented as a Haskell library; the `HCover` tool shared in the latter part of Chapter 4; and the `KCover` tool that is embedded in the `KReach` software that is the focus of Chapter 5. Each of these implementations is available online[1,2,3] and usable as a coverability checker in its own right.

However, only in `HCover` have meaningful steps been taken to improve performance characteristics; it is the only checker of the three that is intentionally competing against other state-of-the-art coverability checkers such as `ICover` and `MIST`. And as Section 4.5 intimates, there is still low-hanging fruit even there for meaningful performance gains, especially for certain classes of nets. Using a prime representation for vectors looks to be particularly exciting, if the indicated improvements are borne out in practice. The sample case used in Section 4.5 is non-ideal in some sense as it induces very large vectors; very sparse nets, especially 1-safe nets, whose prime representation is small, are liable to see significantly greater performance gains. There may certainly be

[1]`https://github.com/dixonary/hcover`
[2]`https://github.com/dixonary/karp-miller`
[3]`https://github.com/dixonary/kosaraju`

scope for a fourth coverability checker—a `VCover`, perhaps—implementing a prime vector representation, possibly with some other improvements derived from the literature.

Karp's and Miller's tree construction for generating the minimal coverability set has been iterated on and improved [144] since its original presentation [87]. Finkel and others have constructed a way of identifying the *clover*, a unique canonical representation of the coverability graph for any Petri net starting from some marking [50, 51]. This representation is used to build out a suite of "commodified" accelerations for the general Karp and Miller construction [55]. The latter work included a tool for generating this minimal coverability graph and testing it against a number of alternatives; results indicate that it is competitive. Implementing that approach, and in doing so making use of some of the natural benefits of working with a compiled, strictly typed language, may prove valuable; alternatively it may be possible to replace the unoptimised Karp and Miller procedure in Chapter 5 by calling out directly to the implementation of Finkel, Haddad and Khmelnitsky.

Another potential avenue of research, as discussed at the end of Chapter 4, is into the ways that we can exploit the properties of modern computing architectures to speed-up the so-called "wall clock time" taken for coverability procedures. Theoretical improvements for these procedures from the literature tend to be in favour of minimising the number of nodes visited in the given search space. This minimises the total *CPU time* taken. However, most modern CPUs have eight, twelve, or more cores which might be used at the same time. Tools implemented in Python are rarely able to make use of these since the Python interpreter is fundamentally non-parallel; `MIST` (the only readily accessible such tool not implemented in Python) appears also to be single-threaded.

In theory, any state-space exploration should be conducive to parallelism. The problem arises when one wishes to prune the exploration; this requires access to a shared list of visited nodes. If the time taken accessing this list can be minimised, then meaningful speedup might still be in reach. If the exploration is not driven by pruning, then parallelising is unproblematic. This is particularly true in Haskell, where data parallelism is lightweight [84].

Beyond the CPU, most modern desktop and notebook personal computers come with a discrete graphics card; such cards modernly have dozens or even hundreds of cores. The global leaders in retail GPU manufacturing, NVIDIA [72] and more recently AMD [139], and open-source computing research and engineering collectives [94, 100] have made ubiquitous the notion of a *GPGPU*: a GPU which can be used for general-purpose computing. Given their common use case of graphics manipulation, GPUs make an excellent candidate for accelerated vector mathematics. A propensity of libraries [27] now exist to make this accessible to the Haskell programmer, and a tool built around them would likely be of interest in its own right (to the practical solving community, or to Haskell programmers, or to those interested in applications of general-purpose GPU computing).

10.2 Petri Net Reachability

The reachability checker KReach is, as far as the authors can tell, the first complete implementation of any algorithm for the reachability problem on Petri nets. Of course, we now know thanks to the diligent work of Czerwiński and Orlikowski [30] and Leroux [102] and others that the reachability problem is complete for ACKERMANN, and so theoretical improvements will only be able to get us so far. Recent results have been primarily concerned with how practically intractable reachability is; we hope that the work of Chapter 5 is some indication that optimism in the world of reachability is not all lost.

That Kosaraju's alogrithm can statically rule out reachability in certain cases derived from coverability instances warrants further attention. Notwithstanding decompositions based on θ, Kosaraju's alogrithm represents the computation of some complex static properties of a VASS, some of which are not dissimilar to the inducive invariants of Blondin et al. (see Section 30). It is possible that the conjunction of these properties makes for a robust set of invariants which could be abstracted out to speed up non-reachability based coverability algorithms.

Furthermore, testing revealed that KReach responded in one of four ways to any given nets: either it immediately fulfilled the θ condition; or it was immediately deduced that θ was unsatisfiable for that net; or it required a small number of decompositions; or the procedure timed out. Trying to classify which nets fall into which category may reveal some new structural properties of those classes, which may offer insight into the types of nets on which reachability can be easily decided. For example, consider a Petri net in which the total number of tokens never increases. If the final marking has more tokens than the initial marking, the θ_1 condition (Section 44) immediately rules out reachability using only the state equation.

Very recently, and further to the work of Amat, Dal Zilio and Hujsa [6] and Blondin, Haase and Offtermatt [15], some (as yet unpublished) work by others tries to solve the reachability problem by pragmatic means, including the use of machine learning and artificial intelligence. One may view this as an automatic form of the classification process described above. Since most nets encountered on a regular basis will *not* be of the algorithmically-catastrophic variety of those seen in proofs of lower bounds [30, 31, 102], such an approach may be workable for nets derived from real-world problems. It seems unlikely that such a model would be good at identifying nonreachable instances, as determining nonreachability requires exhaustive search. (A probabilistic approach may, however, respond with some *confidence* whether or not a solution exists.)

Petri Puzzle

One may ask: if artificial intelligence might offer a way to solve reachability queries, can "non-artifical" intelligence do the same? Computationally hard problems often make for good puzzles. A famous collection of Simon Tatham[4] includes 40 such puzzles, many of which are recognisable as NP-complete problems[5]. In the same spirit, we undertook to implement the Petri net reachability problem as a puzzle. Other than fun, the goals of this were twofold: firstly to determine how difficult solving simple Petri net reachability problems *feels* in practice; and secondly to give people a practical and hands-on way to learn about the formalism.

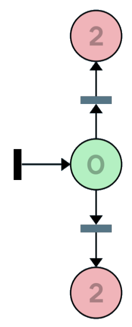
The preliminary version of Petri Puzzle includes seven reachability queries as given in Figure 10.1. The representation in the program is almost exactly that described in Section 3.5.2 and used throughout this thesis. Some conveniences are included to help users to become acquainted with the semantics. For example, a place turns green if the number of tokens in a marking is equal to the number in the target marking for that place; transitions go a lighter colour if they are not firable in the current marking. Each query shows a different feature of the model, including sources, sinks, non-unit arcs, and so on. Extending the format to more exotic variants, such as Petri nets with reset arcs or post-self-modifying nets, would be simple.

The Petri Puzzle program includes a Petri net designer in which one can plan out nets of the type included in the main game. Moreover, Petri Puzzle is a use case for `KReach`: When designing such nets, the program will call out to `KReach` to determine whether the puzzle has a solution (that is, whether the goal is reachable from the initial marking). It does the same thing during play: if `KReach` returns that the player has found a marking from which the goal marking is unreachable, the game will instruct the player to restart the game. Since all these nets are small, `KReach` is able to perform this check in real-time.

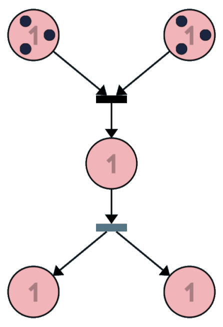
Petri Puzzle has not yet been released, but it may be of interest to the Petri net community and possibly to educators in formal methods.

[4]`https://www.chiark.greenend.org.uk/~sgtatham/puzzles/`

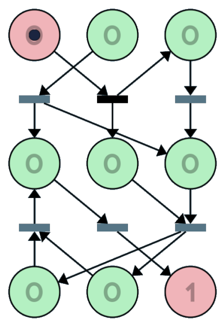
[5]A recreational pastime of a particular flavour of computer scientist is to pick a puzzle from the list and try to determine its algorithmic complexity.



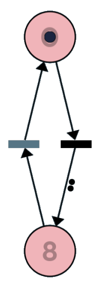
(a) “One”



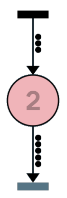
(b) “Multi”



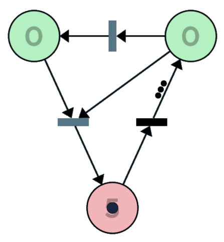
(c) “Trace”



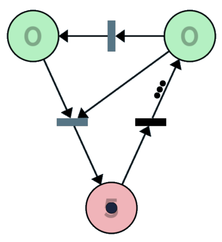
(d) “Doubler”



(e) “Modular”



(f) “Engine”



(g) “Distribution”

Figure 10.1: The seven levels currently encoded in Petri Puzzle. The initial marking is as given; the target marking is the label in each place.

10.3 Leafy Automata, Local Leafy Automata, Split Automata, and Beyond

The work in the second half of this thesis introduces a family of ways to unify game semantics and automata theory in the context of programming languages with concurrency. Each of the formalisms introduced (LA, LLA, SA) makes use of the same underlying structure: a tree configuration backed by a data forest, which pegs a relationship between the game-semantic interpretation of plays of terms and the operational semantics of those terms. As the works progress, it is possible to spot a trend: at each stage we reduce the amount of communication that takes place in the tree in some meaningful way, and in doing so permit a wider class of constructs to exist within the language. In leafy automata (Chapter 6), all constructs of Finitary Idealized Concurrent Algol are permitted, but all decision problems are trivially undecidable. In local leafy automata (Chapter 8), we require that variables are “local”, i.e. that variables may not pass through two calls to free functions. In doing so, we recover decidability for may-termination of `FICA` terms, but we must exclude unbounded iteration (**while**). In split automata (Chapter 9), we allow for only minimal communication in the tree through control states, and instead mandate that communication occur through shared memory. The only constraints placed on that memory is that reads and writes may not occur together atomically (corresponding to bounding the applicative depth of semaphores). Via SA we get that may-termination of FICA terms obeying this semaphore restriction is decidable, and *all* language constructs are permitted including **while**.

A number of open questions remain about the complexity of other decision problems over LLA and SA. In introducing and exploring these models we focused on the emptiness problem, which is sufficient to encode a variety of interesting properties. The equivalence problem is also a potential source of value: if equivalence turns out to be decidable in either setting, that may open the door to checking terms for contextual equivalence. As noted before, there are some challenges present with checking for contextual equivalence due to the representation we use for game-semantic plays—there are a plurality of encodings that may represent the same term, and therefore comparing terms is not quite the same as comparing LLA/SAs for language equivalence. Hence some additional work would need to go into producing some canonical encoding for a given term and, ideally, enforcing that canon in the automaton.

As a model of concurrency, there exists plenty of scope to further situate these tree-configuration-based models within the field of concurrency research. For example, we might seek to apply the concurrent games framework [25] for the sake of verification, or investigate contextual equivalence with respect to semaphore-free contexts [117]. It would also be interesting to look for connections with abstract machines [57], the geometry of interaction [91], and the π-calculus [12].

From the game semantics perspective, there are still some properties of our translation that could be improved. For example, the game semantics includes a natural notion of interleavings: closure under certain rearrangements of moves. An ideal encoding of the game semantics would represent these interleavings faithfully. A new formalism called *saturating automata* has been devised that does precisely that—they can encode this saturation condition while forbidding languages that do not obey it. Saturating automata are another tree-configuration-based automaton and they represent the final step in the trend towards removing control-state-based communicative power between nodes in the data tree: such communication is entirely forbidden. The algorithmic complexity of decision problems over saturating automata remains to be seen, but regardless we hope that they will provide a useful substrate for thinking about game-semantic problems in automata-theoretic terms and possibly act as a vector through which results in automata can be imported to the world of game semantics.

Bibliography

[1] The The Annual Model Checking Competition, 2022. URL `https://mcc.lip6.fr/`.

[2] S Abramsky. Game semantics of idealized parallel algol, 1995.

[3] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation*. Springer-Verlag, 1998. Proceedings of the NATO Advanced Study Institute, Marktoberdorf.

[4] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electron. Notes Theor. Comput. Sci.*, 3:2–14, 1996. doi:10.1016/S1571-0661(05)80398-6.

[5] Rui Alves, Fernando Antunes, and Armindo Salvador. Tools for kinetic modeling of biochemical networks. *Nature biotechnology*, 24(6):667–672, 2006.

[6] Nicolas Amat, Silvano Dal Zilio, and Thomas Hujsa. Property directed reachability for generalized petri nets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 505–523. Springer, 2022.

[7] Elvio Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal Zilio, Francesco Gallà, Lom Messan Hillah, Francis Hulin-Hubard, Peter Gjøl Jensen, Loïg Jezequel, Fabrice Kordon, et al. Presentation of the 9th edition of the model checking contest. In *Proceedings of TACAS*, pages 50–68. Springer, 2019.

[8] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.

[9] John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the 1st International Conference on Information Processing*, pages 125–131. UNESCO (Paris), 1959.

[10] Paolo Baldan, Nicoletta Cocco, Andrea Marin, and Marta Simeoni. Petri nets for modelling metabolic pathways: a survey. *Natural Computing*, 9 (4):955–989, 2010.

[11] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. ISBN 978-0-521-76614-2.

[12] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the pi-calculus. In *Proceedings of TLCA*, volume 2044 of *LNCS*, pages 29–45. Schloss Dagstuhl, 2001. doi:10.1007/3-540-45413-6_7.

[13] Michael Blondin, Alain Finkel, Christoph Haase, and Serge Haddad. Approaching the coverability problem continuously. In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS*, volume 9636 of *LNCS*, pages 480–496. Springer, 2016. doi:10.1007/978-3-662-49674-9_28.

[14] Michael Blondin, Alain Finkel, and Jean Goubault-Larrecq. Forward analysis for WSTS, part III: Karp-Miller trees. *Log. Methods Comput. Sci.*, 16(2), 2020. doi:10.23638/LMCS-16(2:13)2020.

[15] Michael Blondin, Christoph Haase, and Philip Offtermatt. Directed reachability for infinite-state systems. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–23, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72013-1.

[16] Conrad Bock. UML 2 activity modeling for domain experts, 2005. URL `http://conradbock.org/omg/pm/05-12-05.pdf`.

[17] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3):13:1–13:48, 2009. doi:10.1145/1516512.1516515. URL `https://doi.org/10.1145/1516512.1516515`.

[18] Ahmed Bouajjani and Michael Emmi. Analysis of recursively parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(3), nov 2013. ISSN 0164-0925. doi:10.1145/2518188.

[19] Z. Bouziane. A primitive recursive algorithm for the general Petri net reachability problem. In *Proceedings of FOCS*, pages 130–136, 1998. doi:10.1109/SFCS.1998.743436.

[20] Laura Bozzelli and Pierre Ganty. Complexity analysis of the backward coverability algorithm for VASS. In *Proceedings of RP*, volume 6945 of *LNCS*, pages 96–109. Springer, 2011. doi:10.1007/978-3-642-24288-5_10.

[21] Stephen D. Brookes. The essence of parallel algol. *Inf. Comput.*, 179(1): 118–149, 2002. doi:10.1006/inco.2002.2995.

[22] Frank P. Burns, Albert Koelmans, and Alexandre Yakovlev. WCET analysis of superscalar processors using simulation with coloured petri nets. *Real Time Syst.*, 18(2/3):275–288, 2000. doi:10.1023/A:1008101416758.

[23] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[24] E. Cardoza, Richard J. Lipton, and Albert R. Meyer. Exponential space complete problems for petri nets and commutative semigroups: Preliminary report. In *Proceedings of STOC*, pages 50–54. ACM, 1976. doi:10.1145/800113.803630.

[25] S. Castellan, P. Clairambault, S. Rideau, and G. Winskel. Games and strategies as event structures. *Log. Meth. Comput. Sci.*, 13(3), 2017. doi:10.23638/LMCS-13(3:35)2017.

[26] Graziana Cavone, Mariagrazia Dotoli, and Carla Seatzu. A survey on petri net models for freight logistics and transportation systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(6):1795–1813, 2017.

[27] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the POPL Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 3–14. ACM, 2011. doi:10.1145/1926354.1926358.

[28] Lorenzo Clemente, Slawomir Lasota, Ranko Lazic, and Filip Mazowiecki. Binary reachability of timed-register pushdown automata and branching vector addition systems. *ACM Trans. Comput. Log.*, 20(3):14:1–14:31, 2019. doi:10.1145/3326161.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[30] Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *Proceedings of FOCS*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.

[31] Wojciech Czerwiński, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. *J. ACM*, 68(1):7:1–7:28, 2021. doi:10.1145/3422822.

[32] Stéphane Demri, Marcin Jurdziński, Oded Lachish, and Ranko Lazić. The covering and boundedness problems for branching vector addition systems. *J. Comput. Syst. Sci.*, 79(1):23–38, 2013. doi:10.1016/j.jcss.2012.04.002.

[33] Stéphane Demri, Diego Figueira, and M. Praveen. Reasoning about data repetitions with counter systems. *Logical Methods in Computer Science*, 12(3), 2016. doi:10.2168/LMCS-12(3:1)2016.

[34] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979. doi:10.1145/359138.359142.

[35] Ankush Desai, Shromona Ghosh, Sanjit A. Seshia, Natarajan Shankar, and Ashish Tiwari. SOTER: programming safe robotics system using runtime assurance. *CoRR*, abs/1808.07921, 2018. URL `http://arxiv.org/abs/1808.07921`.

[36] Alin Deutsch, Yuliang Li, and Victor Vianu. VERIFAS: A practical verifier for artifact systems. *CoRR*, abs/1705.10007, 2017. URL `http://arxiv.org/abs/1705.10007`.

[37] Gregorio Díaz, Hermenegilda Macià, Valentín Valero, Juan Boubeta-Puig, and Fernando Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored petri nets. *Neural Comput. Appl.*, 32(2):405–426, 2020. doi:10.1007/s00521-018-3850-1.

[38] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422, 1913. ISSN 00029327, 10806377.

[39] A. Dimovski and R. Lazic. CSP representation of game semantics for second-order idealized Algol. In *Proceedings of ICFEM*, LNCS, pages 146–161, 2004.

[40] Alex Dixon and Ranko Lazic. KReach: A tool for reachability in Petri nets. In *Proceedings of TACAS*, volume 12078 of *LNCS*, pages 405–412. Springer, 2020. doi:10.1007/978-3-030-45190-5_22.

[41] Alex Dixon, Ranko Lazic, Andrzej S. Murawski, and Igor Walukiewicz. Leafy automata for higher-order concurrency. In *Proceedings of FOSSACS*, volume 12650 of *LNCS*, pages 184–204. Springer, 2021. doi:10.1007/978-3-030-71995-1_10.

[42] Alex Dixon, Ranko Lazic, Andrzej S. Murawski, and Igor Walukiewicz. Verifying higher-order concurrency with data automata. In *Proceedings of LICS*. IEEE, 2021. doi:10.1109/LICS52264.2021.9470691.

[43] Mariagrazia Dotoli, Nicola Epicoco, Marco Falagario, and Graziana Cavone. A timed Petri nets model for intermodal freight transport terminals. *Proceedings of IFAC*, 47(2):176–181, 2014. ISSN 1474-6670. doi:https://doi.org/10.3182/20140514-3-FR-4046.00038.

[44] Javier Esparza. *Decidability and complexity of Petri net problems—an introduction*, pages 374–428. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49442-3. doi:10.1007/3-540-65306-6_20.

[45] Javier Esparza and Stephan Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods Syst. Des.*, 16(2):159–189, 2000. doi:10.1023/A:1008743212620.

[46] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - a survey. *Bull. EATCS*, 52:244–262, 1994.

[47] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Niksic. An SMT-based approach to coverability analysis. In *Proceedings of CAV*, volume 8559 of *LNCS*, pages 603–619. Springer, 2014. doi:10.1007/978-3-319-08867-9_40.

[48] Javier Esparza, Pierre Ganty, Jérôme Leroux, and Rupak Majumdar. Verification of population protocols. *Acta Inf.*, 54(2):191–215, 2017. doi:10.1007/s00236-016-0272-3.

[49] Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermannian and primitive-recursive bounds with Dickson's lemma. In *Proceedings of LICS*, pages 269–278, 2011. doi:10.1109/LICS.2011.39.

[50] Alain Finkel. The minimal coverability graph for Petri nets. In *Proceedings of PETRI NETS*, volume 674 of *LNCS*, pages 210–243. Springer, 1991. doi:10.1007/3-540-56689-9_45.

[51] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for WSTS, part II: complete WSTS. *Log. Methods Comput. Sci.*, 8(3), 2012. doi:10.2168/LMCS-8(3:28)2012.

[52] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001. doi:10.1016/S0304-3975(00)00102-X.

[53] Alain Finkel, Pierre McKenzie, and Claudine Picaronny. A well-structured framework for analysing petri net extensions. *Inf. Comput.*, 195(1-2):1–29, 2004. doi:10.1016/j.ic.2004.01.005.

[54] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Minimal coverability tree construction made complete and efficient. In *Proceedings of FOSSACS*, volume 12077 of *LNCS*, pages 237–256. Springer, 2020. doi:10.1007/978-3-030-45231-5_13.

[55] Alain Finkel, Serge Haddad, and Igor Khmelnitsky. Commodification of accelerations for the karp and miller construction. *Discret. Event Dyn. Syst.*, 31(2):251–270, 2021. doi:10.1007/s10626-020-00331-z.

[56] Estíbaliz Fraca and Serge Haddad. Complexity analysis of continuous petri nets. *Fundam. Informaticae*, 137(1):1–28, 2015. doi:10.3233/FI-2015-1168.

[57] O. Fredriksson and D. R. Ghica. Abstract machines for game semantics, revisited. In *Proceedings of LICS*, pages 560–569, 2013. doi:10.1109/LICS.2013.63.

[58] Pierre Ganty, Cédric Meuter, Giorgio Delzanno, Gabriel Kalyon, Jean-François Raskin, and Laurent van Begin. Symbolic data structure for sets of k-uples of integers. Technical report, Université Libre de Bruxelles, 2007.

[59] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: Automatic abstraction refinement for petri nets. *Fundam. Informaticae*, 88(3):275–305, 2008. doi:10.1007/978-3-540-73094-1_10.

[60] Pierre Ganty, Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Le problème de couverture pour les réseaux de petri. résultats classiques et développements récents. *Tech. Sci. Informatiques*, 28(9): 1107–1142, 2009. doi:10.3166/tsi.28.1107-1142.

[61] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for petri nets. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proceedings of ATVA*, volume 4762 of *LNCS*, pages 98–113. Springer, 2007. doi:10.1007/978-3-540-75596-8_9.

[62] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set of petri nets. *Int. J. Found. Comput. Sci.*, 21(2):135–165, 2010. doi:10.1142/S0129054110007180.

[63] Thomas Geffroy, Jérôme Leroux, and Grégoire Sutre. Occam's razor applied to the petri net coverability problem. *Theor. Comput. Sci.*, 750: 38–52, 2018. doi:10.1016/j.tcs.2018.04.014.

[64] Vijay Gehlot. From petri nets to colored petri nets: A tutorial introduction to nets based formalism for modeling and simulation. In *Proceedings of the Winter Simulation Conference*, WSC '19, page 1519–1533. IEEE Press, 2019. ISBN 9781728132839.

[65] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. doi:10.1145/146637.146681.

[66] D. R. Ghica and G. McCusker. Reasoning about Idealized Algol using regular expressions. In *Proceedings of ICALP*, volume 1853 of *LNCS*, pages 103–115. Springer-Verlag, 2000.

[67] D. R. Ghica and G. McCusker. The regular language semantics of second-order Idealized Algol. *Theoretical Computer Science*, 309:469–502, 2003.

[68] D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In *Proceedings of TACAS*, volume 3920 of *LNCS*, pages 303–317. Springer, 2006.

[69] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theor. Comp. Sci.*, pages 234–251, 2006. doi:10.1016/j.tcs.2005.10.032.

[70] Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Log.*, 151(2-3):89–114, 2008. doi:10.1016/j.apal.2007.10.005.

[71] Dan R Ghica, Andrzej S Murawski, and C-HL Ong. Syntactic control of concurrency. *Theoretical Computer Science*, 350(2-3):234–251, 2006.

[72] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. GPGPU processing in CUDA architecture. *CoRR*, abs/1202.4347, 2012. URL `http://arxiv.org/abs/1202.4347`.

[73] A.N. Gorban and G.S. Yablonsky. Extended detailed balance for systems with irreversible reactions. *Chemical Engineering Science*, 66(21):5388–5399, nov 2011. doi:10.1016/j.ces.2011.07.054.

[74] Alexander Grosskopf, Gero Decker, and Mathias Weske. *The process: business process modeling using BPMN.* Meghan Kiffer Press, 2009.

[75] M. Hack. The recursive equivalence of the reachability problem and the liveness problem for petri nets and vector addition systems. In *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, pages 156–164, Oct 1974. doi:10.1109/SWAT.1974.28.

[76] Russell Harmer and Guy McCusker. A fully abstract game semantics for finite nondeterminism. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 422–430. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782637.

[77] Ralf Hofestädt. A petri net application to model metabolic processes. *Syst. Anal. Model. Simul.*, 16(2):113–122, oct 1994. ISSN 0232-9298.

[78] A.W. Holt and F. Commoner. *Events and Conditions: an approach to the description and analysis of dynamic systems.* Applied Data Research, Inc. 1970. URL `https://books.google.co.uk/books?id=LtZmHAAACAAJ`.

[79] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8: 135–159, 1979. doi:10.1016/0304-3975(79)90041-0.

[80] Matthias Jantzen and Rüdiger Valk. Formal properties of place/transition nets. In *Net Theory and Applications*, pages 165–212. Springer, 1980.

[81] Petr Jančar. Undecidability of bisimilarity for petri nets and some related problems. *Theor. Comp. Sci.*, 148(2):281–301, 1995. ISSN 0304-3975. doi:https://doi.org/10.1016/0304-3975(95)00037-W.

[82] Kurt Jensen. Coloured Petri nets. In *Petri nets: central models and their properties*, pages 248–299. Springer, 1987.

[83] Jérôme Leroux. The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. *Logical Methods in Computer Science*, Volume 6, Issue 3, September 2010. doi:10.2168/LMCS-6(3:22)2010.

[84] Simon L. Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In G. Ramalingam, editor, *Proceedings of APLAS*, volume 5356 of *LNCS*, page 138. Springer, 2008. doi:10.1007/978-3-540-89330-1_10.

[85] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. A widening approach to multithreaded program verification. *ACM Trans. Program. Lang. Syst.*, 36(4), oct 2014. ISSN 0164-0925. doi:10.1145/2629608.

[86] Max I. Kanovich. Petri nets, Horn programs, linear logic and vector games. *Ann. Pure Appl. Logic*, 75(1–2):107–135, 1995. doi:10.1016/0168-0072(94)00060-G.

[87] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.

[88] Stephen C Kleene et al. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956.

[89] Donald E. Knuth. Mathematics and computer science: Coping with finiteness. *Science*, 194(4271):1235–1242, 1976. doi:10.1126/science.194.4271.1235. URL `https://www.science.org/doi/abs/10.1126/science.194.4271.1235`.

[90] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281. ACM, 1982. doi:10.1145/800070.802201.

[91] U. Dal Lago, R. Tanaka, and A. Yoshimizu. The geometry of concurrent interaction: handling multiple ports by way of multiple tokens. In *Proceedings of LICS*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005112.

[92] J. Laird. A game semantics of Idealized CSP. In *Proceedings of MFPS'01*, pages 1–26. Elsevier, 2001. ENTCS, Vol. 45.

[93] J. Laird. Game semantics for higher-order concurrency. In *FSTTCS*, volume 4337 of *LNCS*, pages 417–428, 2006.

[94] Chris Lamb. OpenCL for NVIDIA GPUs. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–24. IEEE, 2009.

[95] J.L. Lambert. A structure to decide reachability in Petri nets. *Theor. Comput. Sci.*, 99(1):79–104, 1992. doi:10.1016/0304-3975(92)90173-D.

[96] Slawomir Lasota. VASS reachability in three steps. *CoRR*, abs/1812.11966, 2018. URL `http://arxiv.org/abs/1812.11966`.

[97] Sławomir Lasota. Improved Ackermannian Lower Bound for the Petri Nets Reachability Problem. In *Proceedings of STACS*, volume 219 of *LIPIcs*, pages 46:1–46:15. Dagstuhl, 2022. ISBN 978-3-95977-222-8. doi:10.4230/LIPIcs.STACS.2022.46.

[98] Ranko Lazić and Sylvain Schmitz. Nonelementary complexities for branching vass, mell, and extensions. *ACM Trans. Comput. Logic*, 16 (3), jun 2015. ISSN 1529-3785. doi:10.1145/2733375.

[99] Ranko Lazic and Patrick Totzke. What makes Petri nets harder to verify: stack or data? In *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *LNCS*, pages 144–161. Springer, 2017. doi:10.1007/978-3-319-51046-0_8.

[100] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of SC*, pages 1–11. IEEE, 2010.

[101] Hélène Leroux, David Andreu, and Karen Godary-Dejean. Handling exceptions in petri net-based digital architecture: From formalism to implementation on FPGAs. *IEEE Transactions on Industrial Informatics*, 11(4):897–906, 2015. doi:10.1109/TII.2015.2435696.

[102] Jérôme Leroux. The reachability problem for Petri nets is not primitive recursive. *CoRR*, abs/2104.12695, 2021. To appear at FOCS 2021. URL `https://arxiv.org/abs/2104.12695`.

[103] Jérôme Leroux and Sylvain Schmitz. Ideal decompositions for vector addition systems (invited talk). In Nicolas Ollinger and Heribert Vollmer, editors, *Proceedings of STACS*, volume 47 of *LIPIcs*, pages 1:1–1:13. Dagstuhl, 2016. doi:10.4230/LIPIcs.STACS.2016.1.

[104] Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *Proceedings of LICS*. IEEE, 2019. doi:10.1109/LICS.2019.8785796.

[105] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual*. 2000 edition.

[106] P. Malacaria and C. Hankin. Generalized flowcharts and games. In *Proceedings of ICALP 1998*, volume 1443 of *LNCS*, pages 363–374. Springer-Verlag, 1998.

[107] Ernst W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984. doi:10.1137/0213029.

[108] Filip Mazowiecki and Michal Pilipczuk. Reachability for bounded branching VASS. In Wan J. Fokkink and Rob van Glabbeek, editors, *Proceedings of CONCUR*, volume 140 of *LIPIcs*, pages 28:1–28:13. Dagstuhl, 2019. doi:10.4230/LIPIcs.CONCUR.2019.28.

[109] Gérard Memmi and Gérard Roucairol. Linear algebra in net theory. In Wilfried Brauer, editor, *Proceedings of the Advanced Course on General*

Net Theory of Processes and Systems, volume 84 of *LNCS*, pages 213–223. Springer, 1979. doi:10.1007/3-540-10001-6_24.

[110] P.M. Merlin. *A Study of the Recoverability of Computing Systems*. University of California, Irvine, 1974. URL `https://escholarship.org/uc/item/1p80c4fg`.

[111] Roland Meyer. A theory of structural stationarity in the *pi*-calculus. *Acta Inf.*, 46(2):87–137, 2009. doi:10.1007/s00236-009-0091-x.

[112] Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977. doi:10.1016/0304-3975(77)90053-6.

[113] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[114] Horst Müller. The reachability problem for VAS. In Grzegorz Rozenberg, Hartmann J. Genrich, and Gérard Roucairol, editors, *Advances in Petri Nets*, volume 188 of *LNCS*, pages 376–391. Springer, 1984. doi:10.1007/3-540-15204-0_21.

[115] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. doi:10.1109/5.24143.

[116] A. S. Murawski. On program equivalence in languages with ground-type references. In *Proceedings of LICS*, pages 108–117. Computer Society Press, 2003.

[117] A. S. Murawski. Full abstraction without synchronization primitives. In *Proceedings of MFPS*, volume 265 of *ENTCS*, pages 423–436, 2010. doi:10.1016/j.entcs.2010.08.025.

[118] A. S. Murawski and N. Tzevelekos. An invitation to game semantics. *SIGLOG News*, 3(2):56–67, 2016.

[119] A. S. Murawski and I. Walukiewicz. Third-order Idealized Algol with iteration is decidable. In *Proceedings of FOSSACS*, volume 3441 of *LNCS*, pages 202–218. Springer, 2005.

[120] Yale University. Department of Computer Science and R.J. Lipton. *The reachability problem requires exponential space*. Research report (Yale University. Department of Computer Science). Department of Computer Science, Yale University, 1976.

[121] C.-H. L. Ong. Observational equivalence of 3rd-order Idealized Algol is decidable. In *Proceedings of LICS*, pages 245–256. Computer Society Press, 2002.

[122] Rohit J Parikh. Language generating devices. *Quarterly Progress Report*, 60:199–212, 1961.

[123] Mor Peleg, Daniel Rubin, and Russ B. Altman. Using Petri Net Tools to Study Properties and Dynamics of Biological Systems. *Journal of the American Medical Informatics Association*, 12(2):181–199, 03 2005. ISSN 1067-5027. doi:10.1197/jamia.M1637.

[124] Elisabeth Pelz, Abderraouf Kabouche, and Louchka Popova-Zeugmann. Interval-timed petri nets with auto-concurrent semantics and their state equation. In *Proceedings of PNSE*, volume 1372 of *CEUR Workshop Proceedings*, pages 245–265, 2015. URL `http://ceur-ws.org/Vol-1372/paper14.pdf`.

[125] Carl Adam Petri. Communication with automata. 1966.

[126] Artturi Piipponen and Antti Valmari. Constructing minimal coverability sets. *Fundam. Informaticae*, 143(3-4):393–414, 2016. doi:10.3233/FI-2016-1319.

[127] Raphaël Plasson. *Chemical Reaction Network*, pages 287–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-11274-4. doi:10.1007/978-3-642-11274-4_269.

[128] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comp. Sci.*, 6(2):223–231, 1978. ISSN 0304-3975. doi:https://doi.org/10.1016/0304-3975(78)90036-1.

[129] Owen Rambow. Multiset-valued linear index grammars: Imposing dominance constraints on derivations. In *Proceedings of the ACL*, pages 263–270. Morgan Kaufmann Publishers, 1994. doi:10.3115/981732.981768.

[130] Venkatramana N Reddy, Michael L Mavrovouniotis, Michael N Liebman, et al. Petri net representations in metabolic pathways. In *ISMB*, volume 93, pages 328–336, 1993.

[131] Venkatramana N Reddy, Michael L Mavrovouniotis, and Michael N Liebman. Modeling biological pathways: a discrete-event systems approach. ACS Publications, 1994.

[132] Claus Reinke. Haskell-coloured Petri nets. In *Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 165–180. Springer, 1999. doi:10.1007/10722298_10.

[133] Wolfgang Reisig. Understanding petri nets modeling techniques, analysis methods, case studies. *Bull. EATCS*, 112, 2014. URL `http://eatcs.org/beatcs/index.php/beatcs/article/view/250`.

[134] Applied Data Research and A.W. Holt. *Information System Theory Project.* Rome Air Development Center. Technical report. Rome Air Development Center, Air Force Systems Command, Griffiss Air Base, 1968. URL `https://books.google.co.uk/books?id=ZkfSNm0WxuQC`.

[135] C. Reutenauer. *The mathematics of Petri nets.* Prentice Hall, 1990. ISBN 9780135618875. Translated by Ian Craig.

[136] Christophe Reutenauer. *Aspects Mathématiques des Réseaux de Petri.* Masson, 1989.

[137] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for petri nets: Karp and miller algorithm with pruning. *Fundam. Informaticae*, 122(1-2):1–30, 2013. doi:10.3233/FI-2013-781.

[138] John C. Reynolds. *The Essence of ALGOL*, page 67–88. Birkhauser Boston Inc., USA, 1997. ISBN 0817638806.

[139] M Sabony. GPUFORT: A source-to-source translator for Fortran accelerator dialects. In *Fifth EAGE Workshop on High Performance Computing for Upstream*, volume 1, pages 1–5. European Association of Geoscientists & Engineers, 2021.

[140] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 61–76. ACM, 1977. doi:10.1145/800105.803396. URL `https://doi.org/10.1145/800105.803396`.

[141] Sylvain Schmitz. Complexity hierarchies beyond elementary. *ACM Trans. Comput. Theory*, 8(1):3:1–3:36, 2016. doi:10.1145/2858784. URL `https://doi.org/10.1145/2858784`.

[142] Manuel Silva, Enrique Terue, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In *Advanced Course on Petri Nets*, pages 309–373. Springer, 1996.

[143] Rüdiger Valk. Self-modifying nets, a natural extension of petri nets. In Giorgio Ausiello and Corrado Böhm, editors, *Proceedings of ALP*, volume 62 of *LNCS*, pages 464–476. Springer, 1978. doi:10.1007/3-540-08860-1_35.

[144] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. *Fundam. Informaticae*, 131(1):1–25, 2014. doi:10.3233/FI-2014-1002.

[145] Wil M. P. van der Aalst. Business process management as the "killer app" for Petri nets. *Softw. Syst. Model.*, 14(2):685–691, 2015. doi:10.1007/s10270-014-0424-2.

[146] Kumar Neeraj Verma. Two-way equational tree automata for ac-like theories: Decidability and closure properties. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *LNCS*, pages 180–196. Springer, 2003. doi:10.1007/3-540-44881-0_14. URL `https://doi.org/10.1007/3-540-44881-0_14`.

[147] Kumar Neeraj Verma and Jean Goubault-Larrecq. Karp-miller trees for a branching extension of VASS. *Discret. Math. Theor. Comput. Sci.*, 7 (1):217–230, 2005. URL `http://dmtcs.episciences.org/350`.

[148] Harro Wimmel and Karsten Wolf. Applying CEGAR to the petri net state equation. *Log. Methods Comput. Sci.*, 8(3), 2012. doi:10.2168/LMCS-8(3:27)2012.

[149] W.M. Zuberek. Timed petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991. ISSN 0026-2714. doi:10.1016/0026-2714(91)90007-T.